TutorialsPoint 后端教程

wizardforcel

Published with GitBook

目錄

W3School 后端教程合集

来源：菜鸟教程

整理：飞龙

感谢菜鸟教程站长的翻译和奉献。

Django 教程

Django 安装

在安装 Django 前，系统需要已经安装了Python的开发环境。接下来我们来具体看下不同系统下Django的安装。

Window 下安装 Django

如果你还未安装Python环境需要先下载Python安装包。

1、Python 下载地址：https://www.python.org/downloads/

2、Django 下载地址：https://www.djangoproject.com/download/

注意：目前Django 1.6.x以上版本已经完全兼容Python 3.x。

Python 安装(已安装的可跳过)

安装Python你只需要下载python-x.x.x.msi文件，然后一直点击"Next"按钮即可。

安装完成后你需要设置Python环境变量。 右击计算机->属性->高级->环境变量->修改系统变量path，添加Python安装地址，本文实例使用的是C:\Python33，你需要根据你实际情况来安装。

Django 安装

下载 Django 压缩包，解压并和Python安装目录放在同一个根目录，进入 Django 目录，执行python setup.py install，然后开始安装，Django将要被安装到Python的Lib下site-packages。

然后是配置环境变量，将这几个目录添加到系统环境变量中： C:/Python33/Lib/site-packages/django;C:/Python33/Scripts。 添加完成后就可以使用Django的django-admin.py命令新建工程了。

检查是否安装成功

输入以下命令进行检查:

```
python import django
django.get_version()
```

如果输出了Django的版本号说明安装正确。

Linux 上安装 Django

yum 安装方法

以下安装位于 Centos Linux 环境下安装，如果是你的 Linux 系统是 ubuntu 请使用 apt-get 命令。

默认情况下 Linux 环境已经支持了Python。你可以在终端输入Python命令来查看是否已经安装。

```
Python  2.7.3  (default,  Aug  1  2012,  05:14:39)  [GCC 4.6.3] on linux2 Type  "help",
```

安装 setuptools

命令：

```
yum install setuptools
```

完成之后，就可以使用 easy_install 命令安装 django

```
easy_install django
```

之后我们在python解释器输入以下代码：

```
[root@solar django]# python Python  2.7.3  (default,  May  15  2014,  14:49:08)  [GCC 4.8
```

我们可以看到输出了Django的版本号，说明安装成功。

pip 命令安装方法

```
pip install Django
```

源码安装方法

下载源码包：https://www.djangoproject.com/download/

输入以下命令并安装：

```
tar xzvf Django-X.Y.tar.gz # 解压下载包 cd Django-X.Y # 进入 Django 目录 python setup.py ins
```

安装成功后 Django 位于 Python 安装目录的 site-packages 目录下。

Mac 下安装

下载

从这里下载最新的稳定版本：DJango-1.x.y.tar.gz，在页面右侧列表下载，如下图：记住是最新的官方版本哦.其中x.y是版本号。进入你下载该文件的文件夹目录，执行如下命令:（Mac下默认是/Users/xxx/Downloads，xxx是你的用户名）

```
$ tar zxvf Django-1.x.y.tar.gz
```

你也可以从 Github 上下载最新版，地址：https://github.com/django/django：

```
git clone https://github.com/django/django.git
```

安装

进入解压后的目录：

```
cd Django-1.x.y
sudo python setup.py install
```

安装成功后会输出以下信息：

```
…… Processing dependencies for  Django==1.x.y Finished processing dependencies for  Djan
```

再进入我们的站点目录，创建 Django 项目：

```
$ django-admin.py startproject testdj
```

启动服务：

```
cd testdj # 切换到我们创建的项目 $ python manage.py runserver ……  Starting development serve
```

以上信息说明，项目已启动，访问地址为http://127.0.0.1:8000/。

Django 创建第一个项目

本章我们将介绍Django 管理工具及如何使用 Django 来创建项目，第一个项目我们以 HelloWorld 来命令项目。

Django 管理工具

安装 Django 之后，您现在应该已经有了可用的管理工具 django-admin.py。我们可以使用 django-admin.py 来创建一个项目:

我们可以来看下django-admin.py的命令介绍:

```
[root@solar ~]# django-admin.py
Usage: django-admin.py subcommand [options] [args]

Options:
  -v VERBOSITY, --verbosity=VERBOSITY
                        Verbosity level; 0=minimal output, 1=normal output,
                        2=verbose output, 3=very verbose output
  --settings=SETTINGS   The Python path to a settings module, e.g.
                        "myproject.settings.main". If this isn't provided, the
                        DJANGO_SETTINGS_MODULE environment variable will be
                        used.
  --pythonpath=PYTHONPATH
                        A directory to add to the Python path, e.g.
                        "/home/djangoprojects/myproject".
  --traceback           Raise on exception
  --version             show program's version number and exit
  -h, --help            show this help message and exit

Type 'django-admin.py help <subcommand>' for help on a specific subcommand.

Available subcommands:

[django]
    check
    cleanup
    compilemessages
    createcachetable
……省略部分……
```

创建第一个项目

使用 django-admin.py 来创建 HelloWorld 项目：

```
django-admin.py startproject HelloWorld
```

创建完成后我们可以查看下项目的目录结构：

```
[root@solar ~]# cd HelloWorld/
[root@solar HelloWorld]# tree
.
|-- HelloWorld
|   |-- __init__.py
|   |-- settings.py
|   |-- urls.py
|   `-- wsgi.py
`-- manage.py
```

目录说明：

- **HelloWorld:** 项目的容器。
- **manage.py:** 一个实用的命令行工具，可让你以各种方式与该 Django 项目进行交互。
- **HelloWorld/init.py:** 一个空文件，告诉 Python 该目录是一个 Python 包。
- **HelloWorld/settings.py:** 该 Django 项目的设置/配置。
- **HelloWorld/urls.py:** 该 Django 项目的 URL 声明; 一份由 Django 驱动的网站"目录"。
- **HelloWorld/wsgi.py:** 一个 WSGI 兼容的 Web 服务器的入口，以便运行你的项目。

接下来我们进入 HelloWorld 目录输入以下命令，启动服务器：

```
python manage.py runserver 0.0.0.0:8000
```

0.0.0.0让其它电脑可连接到开发服务器，8000为端口号。如果不说明，那么端口号默认为8000。

在浏览器输入你服务器的ip及端口号，如果正常启动，输出结果如下：

视图和 URL 配置

在先前创建的 HelloWorld 目录下的 HelloWorld 目录新建一个 view.py 文件，并输入代码：

```
from django.http import HttpResponse

def hello(request):
    return HttpResponse("Hello world ! ")
```

接着，绑定 URL 与视图函数。打开 urls.py 文件，删除原来代码，将以下代码复制粘贴到 urls.py 文件中：

```
from django.conf.urls import *
from HelloWorld.view import hello

urlpatterns = patterns("",
    ('^hello/$', hello),
)
```

整个目录结构如下：

```
[root@solar HelloWorld]# tree
.
|-- HelloWorld
|   |-- __init__.py
|   |-- __init__.pyc
|   |-- settings.py
|   |-- settings.pyc
|   |-- urls.py              # url 配置
|   |-- urls.pyc
|   |-- view.py              # 添加的视图文件
|   |-- view.pyc             # 编译后的视图文件
|   |-- wsgi.py
|   `-- wsgi.pyc
`-- manage.py
```

完成后，启动 Django 开发服务器，并在浏览器访问打开浏览器并访问：

注意：项目中如果代码有改动，服务器会自动监测代码的改动并自动重新载入，所以如果你已经启动了服务器则不需手动重启。

Django 模板

在上一章节中我们使用 django.http.HttpResponse() 来输出"Hello World！"。该方式将数据与视图混合在一起，不符合Django的MVC思想。

本章节我们将为大家详细介绍 Django 模板的应用，模板是一个文本，用于分离文档的表现形式和内容。

模板应用实例

我们接着上一章节的项目将在 HelloWorld 目录底下创建 templates 目录并建立 hello.html文件，整个目录结构如下：

```
HelloWorld/
|-- HelloWorld
|   |-- __init__.py
|   |-- __init__.pyc
|   |-- settings.py
|   |-- settings.pyc
|   |-- urls.py
|   |-- urls.pyc
|   |-- view.py
|   |-- view.pyc
|   |-- wsgi.py
|   `-- wsgi.pyc
|-- manage.py
`-- templates
    `-- hello.html
```

hello.html 文件代码如下：

```
<h1>{{ hello }}</h1>
```

从模板中我们知道变量使用了双括号。

接下来我们需要向Django说明模板文件的路径，修改HelloWorld/settings.py，修改 TEMPLATES 中的 DIRS 为 **[BASE_DIR+"/templates",]**，如下所示:

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [BASE_DIR+"/templates",],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
            ],
        },
    },
]
```

我们现在修改 view.py，增加一个新的对象，用于向模板提交数据：

```
# -*- coding: utf-8 -*-

#from django.http import HttpResponse
from django.shortcuts import render

def hello(request):
    context          = {}
    context['hello'] = 'Hello World!'
    return render(request, 'hello.html', context)
```

可以看到，我们这里使用render来替代之前使用的HttpResponse。render还使用了一个字典context作为参数。

context 字典中元素的键值 "hello" 对应了模板中的变量 "{{ hello }}"。

再访问访问http://192.168.45.3:8000/hello/，可以看到页面：

这样我们就完成了使用模板来输出数据，从而实现数据与视图分离。

接下来我们将具体介绍模板中常用的语法规则。

Django 模板标签

if/else 标签

基本语法格式如下：

```
{% if condition %}
     ... display
{% endif %}
```

或者：

```
{% if condition1 %}
   ... display 1
{% elif condiiton2 %}
   ... display 2
{% else %}
   ... display 3
{% endif %}
```

根据条件判断是否输出。if/else 支持嵌套。

{% if %} 标签接受 and ， or 或者 not 关键字来对多个变量做判断 ， 或者对变量取反 （ not)，例如：

```
{% if athlete_list and coach_list %}
    athletes 和 coaches 变量都是可用的。
{% endif %}
```

for 标签

{% for %} 允许我们在一个序列上迭代。

与Python的 for 语句的情形类似，循环语法是 for X in Y ， Y是要迭代的序列而X是在每一个特定的循环中使用的变量名称。

每一次循环中，模板系统会渲染在 {% for %} 和 {% endfor %} 之间的所有内容。

例如，给定一个运动员列表 athlete_list 变量，我们可以使用下面的代码来显示这个列表：

```
<ul>
{% for athlete in athlete_list %}
    <li>{{ athlete.name }}</li>
{% endfor %}
</ul>
```

给标签增加一个 reversed 使得该列表被反向迭代：

```
{% for athlete in athlete_list reversed %}
...
{% endfor %}
```

可以嵌套使用 {% for %} 标签：

```
{% for athlete in athlete_list %}
    <h1>{{ athlete.name }}</h1>
    <ul>
    {% for sport in athlete.sports_played %}
        <li>{{ sport }}</li>
    {% endfor %}
    </ul>
{% endfor %}
```

ifequal/ifnotequal 标签

{% ifequal %} 标签比较两个值，当他们相等时，显示在 {% ifequal %} 和 {% endifequal %} 之中所有的值。

下面的例子比较两个模板变量 user 和 currentuser :

```
{% ifequal user currentuser %}
    <h1>Welcome!</h1>
{% endifequal %}
```

和 {% if %} 类似， {% ifequal %} 支持可选的 {% else%} 标签：8

```
{% ifequal section 'sitenews' %}
    <h1>Site News</h1>
{% else %}
    <h1>No News Here</h1>
{% endifequal %}
```

注释标签

Django 注释使用 {# #}。

```
{# 这是一个注释 #}
```

过滤器

模板过滤器可以在变量被显示前修改它，过滤器使用管道字符，如下所示：

```
{{ name|lower }}
```

{{ name }} 变量被过滤器 lower 处理后，文档大写转换文本为小写。

过滤管道可以被 套接 ，既是说，一个过滤器管道的输出又可以作为下一个管道的输入：

```
{{ my_list|first|upper }}
```

以上实例将第一个元素并将其转化为大写。

有些过滤器有参数。 过滤器的参数跟随冒号之后并且总是以双引号包含。 例如：

```
{{ bio|truncatewords:"30" }}
```

这个将显示变量 bio 的前30个词。

其他过滤器：

- addslashes : 添加反斜杠到任何反斜杠、单引号或者双引号前面。
- date : 按指定的格式字符串参数格式化 date 或者 datetime 对象，实例：

  ```
  {{ pub_date|date:"F j, Y" }}
  ```

- length : 返回变量的长度。

include 标签

{% include %} 标签允许在模板中包含其它的模板的内容。

下面这两个例子都包含了 nav.html 模板：

```
{% include "nav.html" %}
```

模板继承

模板可以用继承的方式来实现复用。

接下来我们先创建之前项目的 templates 目录中添加 base.html 文件，代码如下：

```
<html>
  <head>
    <title>Hello World!</title>
  </head>

  <body>
    <h1>Hello World!</h1>
    {% block mainbody %}
       <p>original</p>
    {% endblock %}
  </body>
</html>
```

以上代码中，名为mainbody的block标签是可以被继承者们替换掉的部分。

所有的 {% block %} 标签告诉模板引擎，子模板可以重载这些部分。

hello.html中继承base.html，并替换特定block，hello.html修改后的代码如下：

```
{% extends "base.html" %}

{% block mainbody %}
<p>继承了 base.html 文件</p>
{% endblock %}
```

第一行代码说明hello.html继承了 base.html 文件。可以看到，这里相同名字的block标签用以替换base.html的相应block。

重新访问地址http://192.168.45.3:8000/hello/，输出结果如下：

Django 模型

Django 对各种数据库提供了很好的支持，包括：PostgreSQL、MySQL、SQLite、Oracle。

Django 为这些数据库提供了统一的调用API。 我们可以根据自己业务需求选择不同的数据库。

MySQL 是 Web 应用中最常用的数据库。本章节我们将以 Mysql 作为实例进行介绍。你可以通过本站的 MySQL 教程 了解更多Mysql的基础知识。

数据库配置

我们在项目的 settings.py 文件中找到 DATABASES 配置项，将其信息修改为：

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'test',
        'USER': 'test',
        'PASSWORD': 'test123',
        'HOST':'localhost',
        'PORT':'3306',
    }
}
```

上面包含数据库名称和用户的信息，它们与MySQL中对应数据库和用户的设置相同。Django根据这一设置，与MySQL中相应的数据库和用户连接起来。

定义模型

创建 APP

Django规定，如果要使用模型，必须要创建一个app。我们使用以下命令创建一个 TestModel 的 app:

```
python manage.py startapp TestModel
```

目录结构如下：

```
HelloWorld
|-- TestModel
|   |-- __init__.py
|   |-- admin.py
|   |-- models.py
|   |-- tests.py
|   `-- views.py
```

我们修改 TestModel/models.py文件，代码如下：

```
# models.py
from django.db import models

class Test(models.Model):
    name = models.CharField(max_length=20)
```

以上的类名代表了数据库表名，且继承了models.Model，类里面的字段代表数据表中的字段(name)，数据类型则由CharField（相当于varchar）、DateField（相当于datetime），max_length 参数限定长度。

接下来在settings.py中找到INSTALLED_APPS这一项，如下：

```
INSTALLED_APPS = (
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'TestModel',               # 添加此项
)
```

在命令行中运行python manage.py syncdb，看到几行"Creating table…"的字样，你的数据表就创建好了。

```
Creating tables ...
……
Creating table TestModel_test  #我们自定义的表
……
```

表名组成结构为：app名_类名（如：TestModel_test）。

注意：尽管我们没有在models给表设置主键，但是Django会自动添加一个id作为主键。

数据库操作

接下来我们在 HelloWorld 目录中添加 testdb.py 文件，并修改urls.py：

```
from django.conf.urls import *
from HelloWorld.view import hello
from HelloWorld.testdb import testdb

urlpatterns = patterns("",
        ('^hello/$', hello),
        ('^testdb/$', testdb),
)
```

添加数据

添加数据需要先创建对象，然后再执行 save 函数，相当于SQL中的INSERT：

```
# -*- coding: utf-8 -*-

from django.http import HttpResponse

from TestModel.models import Test

# 数据库操作
def testdb(request):
    test1 = Test(name='w3cschool.cc')
    test1.save()
    return HttpResponse("<p>数据添加成功！</p>")
```

访问http://192.168.45.3:8000/testdb/ 就可以看到数据添加成功的提示。

获取数据

Django提供了多种方式来获取数据库的内容，如下代码所示：

```
# -*- coding: utf-8 -*-

from django.http import HttpResponse

from TestModel.models import Test

# 数据库操作
def testdb(request):
    # 初始化
    response = ""
    response1 = ""

    # 通过objects这个模型管理器的all()获得所有数据行，相当于SQL中的SELECT * FROM
    list = Test.objects.all()

    # filter相当于SQL中的WHERE，可设置条件过滤结果
    response2 = Test.objects.filter(id=1)

    # 获取单个对象
    response3 = Test.objects.get(id=1)

    # 限制返回的数据 相当于 SQL 中的 OFFSET 0 LIMIT 2;
    Test.objects.order_by('name')[0:2]

    #数据排序
    Test.objects.order_by("id")

    # 上面的方法可以连锁使用
    Test.objects.filter(name="w3cschool.cc").order_by("id")

    # 输出所有数据
    for var in list:
        response1 += var.name + " "
    response = response1
    return HttpResponse("<p>" + response + "</p>")
```

输出结果如下图所示：

更新数据

修改数据可以使用 save() 或 update():

```
# -*- coding: utf-8 -*-

from django.http import HttpResponse

from TestModel.models import Test

# 数据库操作
def testdb(request):
    # 修改其中一个id=1的name字段，再save，相当于SQL中的UPDATE
    test1 = Test.objects.get(id=1)
    test1.name = 'w3cschool菜鸟教程'
    test1.save()

    # 另外一种方式
    #Test.objects.filter(id=1).update(name='w3cschool菜鸟教程')

    # 修改所有的列
    # Test.objects.all().update(name='w3cschool菜鸟教程')

    return HttpResponse("<p>修改成功</p>")
```

删除数据

删除数据库中的对象只需调用该对象的delete()方法即可：

```
# -*- coding: utf-8 -*-

from django.http import HttpResponse

from TestModel.models import Test

# 数据库操作
def testdb(request):
    # 删除id=1的数据
    test1 = Test.objects.get(id=1)
    test1.delete()

    # 另外一种方式
    # Test.objects.filter(id=1).delete()

    # 删除所有数据
    # Test.objects.all().delete()

    return HttpResponse("<p>删除成功</p>")
```

Django 表单

HTML表单是网站交互性的经典方式。 本章将介绍如何用Django对用户提交的表单数据进行处理。

HTTP 请求

HTTP协议以"请求－回复"的方式工作。客户发送请求时，可以在请求中附加数据。服务器通过解析请求，就可以获得客户传来的数据，并根据URL来提供特定的服务。

GET 方法

我们在之前的项目中创建一个 search.py 文件，用于接收用户的请求：

```
# -*- coding: utf-8 -*-

from django.http import HttpResponse
from django.shortcuts import render_to_response

# 表单
def search_form(request):
    return render_to_response('search_form.html')

# 接收请求数据
def search(request):
    request.encoding='utf-8'
    if 'q' in request.GET:
        message = '你搜索的内容为: ' + request.GET['q'].encode('utf-8')
    else:
        message = '你提交了空表单'
    return HttpResponse(message)
```

在模板目录template中添加 search_form.html 表单：

```
<html>
<head>
    <meta charset="utf-8" />
    <title>Search - w3cschool.cc</title>
</head>
<body>
    <form action="/search/" method="get">
        <input type="text" name="q">
        <input type="submit" value="Search">
    </form>
</body>
</html>
```

urls.py 规则修改为如下形式：

```
from django.conf.urls import *
from HelloWorld.view import hello
from HelloWorld.testdb import testdb
from HelloWorld import search

urlpatterns = patterns("",
    ('^hello/$', hello),
    ('^testdb/$', testdb),
    (r'^search-form/$', search.search_form),
    (r'^search/$', search.search),
)
```

访问地址：http://192.168.45.3:8000/search-form/并搜索，结果如下所示:

POST 方法

上面我们使用了GET方法。视图显示和请求处理分成两个函数处理。

提交数据时更常用POST方法。我们下面使用该方法，并用一个URL和处理函数，同时显示视图和处理请求。

我们在tmplate 创建 post.html：

```
<html>
<head>
    <meta charset="utf-8" />
    <title>Search - w3cschool.cc</title>
</head>
<body>
    <form action="/search-post/" method="post">
        {% csrf_token %}
        <input type="text" name="q">
        <input type="submit" value="Submit">
    </form>

    <p>{{ rlt }}</p>
</body>
</html>
```

在模板的末尾，我们增加一个rlt记号，为表格处理结果预留位置。

表格后面还有一个{% csrf_token %}的标签。csrf全称是Cross Site Request Forgery。这是Django提供的防止伪装提交请求的功能。POST方法提交的表格，必须有此标签。

在HelloWorld目录下新建 search2.py 文件并使用 search_post 函数来处理 POST 请求：

```
# -*- coding: utf-8 -*-

from django.shortcuts import render
from django.core.context_processors import csrf

# 接收POST请求数据
def search_post(request):
    ctx ={}
    ctx.update(csrf(request))
    if request.POST:
        ctx['rlt'] = request.POST['q']
    return render(request, "post.html", ctx)
```

urls.py 规则修改为如下形式：

```
from django.conf.urls import *
from HelloWorld.view import hello
from HelloWorld.testdb import testdb
from HelloWorld import search
from HelloWorld import search2

urlpatterns = patterns("",
    ('^hello/$', hello),
    ('^testdb/$', testdb),
    (r'^search-form/$', search.search_form),
    (r'^search/$', search.search),
    (r'^search-post/$', search2.search_post),
)
```

访问 http://192.168.45.3:8000/search-post/ 显示结果如下：

完成以上实例后，我们的目录结构为：

```
HelloWorld
|-- HelloWorld
|   |-- __init__.py
|   |-- __init__.pyc
|   |-- models.pyc
|   |-- search.py
|   |-- search.pyc
|   |-- search2.py
|   |-- search2.pyc
|   |-- settings.py
|   |-- settings.pyc
|   |-- testdb.py
|   |-- testdb.pyc
|   |-- urls.py
|   |-- urls.pyc
|   |-- view.py
|   |-- view.pyc
|   |-- wsgi.py
|   `-- wsgi.pyc
|-- TestModel
|   |-- __init__.py
|   |-- __init__.pyc
|   |-- admin.py
|   |-- models.py
|   |-- models.pyc
|   |-- tests.py
|   `-- views.py
|-- manage.py
`-- templates
    |-- base.html
    |-- hello.html
    |-- post.html
    `-- search_form.html

3 directories, 29 files
```

Request 对象

每个view函数的第一个参数是一个HttpRequest对象，就像下面这个hello()函数:

```
from django.http import HttpResponse

def hello(request):
    return HttpResponse("Hello world")
```

HttpRequest对象包含当前请求URL的一些信息：

属性	描述
`path`	请求页面的全路径,不包括域名—例如, "/hello/"。
`method`	请求中使用的HTTP方法的字符串表示。全大写表示。例如: `if request.method == 'GET':   do_something()` `elif request.method == 'POST':   do_something_else()`
`GET`	包含所有HTTP GET参数的类字典对象。参见QueryDict 文档。
	包含所有HTTP POST参数的类字典对象。参见QueryDict 文档。服务器收到空的POST请求的情况也是有可能发生的。也就是说，表单form通过HTTP POST方法提交请求，但是表单中可以没有数据。因此，不

`POST`	能使用语句if request.POST来判断是否使用HTTP POST方法；应该使用if request.method == "POST" (参见本表的method属性)。注意: POST不包括file-upload信息。参见FILES属性。
`REQUEST`	为了方便，该属性是POST和GET属性的集合体，但是有特殊性，先查找POST属性，然后再查找GET属性。借鉴PHP's `$_REQUEST` 。例如，如果 `GET = {"name": "john"}` 和 `POST = {"age": '34'}` ,则 `REQUEST["name"]` 的值是"john", `REQUEST["age"]` 的值是"34".强烈建议使用GET and POST,因为这两个属性更加显式化，写出的代码也更易理解。
`COOKIES`	包含所有cookies的标准Python字典对象。Keys和values都是字符串。参见第12章，有关于cookies更详细的讲解。
`FILES`	包含所有上传文件的类字典对象。FILES中的每个Key都是<input type="file" name="" />标签中name属性的值. FILES中的每个value 同时也是一个标准Python字典对象，包含下面三个Keys: `filename` : 上传文件名,用Python字符串表示 `content-type` : 上传文件的Content type `content` : 上传文件的原始内容 注意：只有在请求方法是POST，并且请求页面中<form>有enctype="multipart/form-data"属性时FILES才拥有数据。否则，FILES 是一个空字典。
`META`	包含所有可用HTTP头部信息的字典。 例如: `CONTENT_LENGTH` `CONTENT_TYPE` `QUERY_STRING` : 未解析的原始查询字符串 `REMOTE_ADDR` : 客户端IP地址 `REMOTE_HOST` : 客户端主机名 `SERVER_NAME` : 服务器主机名 `SERVER_PORT` : 服务器端口 META 中这些头加上前缀HTTP_最为Key, 例如: `HTTP_ACCEPT_ENCODING` `HTTP_ACCEPT_LANGUAGE` `HTTP_HOST` : 客户发送的HTTP主机头信息 `HTTP_REFERER` : referring页 `HTTP_USER_AGENT` : 客户端的user-agent字符串 `HTTP_X_BENDER` : X-Bender头信息
`user`	是一个django.contrib.auth.models.User 对象，代表当前登录的用户。如果访问用户当前没有登录，user将被初始化为django.contrib.auth.models.AnonymousUser的实例。你可以通过user的is_authenticated()方法来辨别用户是否登录： `if request.user.is_authenticated():` `# Do something for logged-in users.` `else:` `# Do something for anonymous users.` 只有激活Django中的AuthenticationMiddleware时该属性才可用
`session`	唯一可读写的属性，代表当前会话的字典对象。只有激活Django中的session支持时该属性才可用。 参见第12章。
`raw_post_data`	原始HTTP POST数据，未解析过。 高级处理时会有用处。

Request对象也有一些有用的方法：

方法	描述
getitem(key)	返回GET/POST的键值,先取POST,后取GET。如果键不存在抛出KeyError。这是我们可以使用字典语法访问HttpRequest对象。例如,request["foo"]等同于先request.POST["foo"] 然后request.GET["foo"]的操作。
has_key()	检查request.GET or request.POST中是否包含参数指定的Key。
get_full_path()	返回包含查询字符串的请求路径。例如， "/music/bands/the_beatles/?print=true"
is_secure()	如果请求是安全的，返回True，就是说，发出的是HTTPS请求。

QueryDict对象

在HttpRequest对象中, GET和POST属性是django.http.QueryDict类的实例。

QueryDict类似字典的自定义类，用来处理单键对应多值的情况。

QueryDict实现所有标准的词典方法。还包括一些特有的方法：

方法	描述
`__getitem__`	和标准字典的处理有一点不同，就是，如果Key对应多个Value，**getitem**()返回最后一个value。
`__setitem__`	设置参数指定key的value列表(一个Python list)。注意：它只能在一个mutable QueryDict 对象上被调用(就是通过copy()产生的一个QueryDict对象的拷贝).
`get()`	如果key对应多个value，get()返回最后一个value。
`update()`	参数可以是QueryDict，也可以是标准字典。和标准字典的update方法不同，该方法添加字典 items，而不是替换它们: `>>> q = QueryDict('a=1')` `>>> q = q.copy() # to make it mutable` `>>> q.update({'a': '2'})` `>>> q.getlist('a')` `['1', '2']` `>>> q['a'] # returns the last` `['2']`
`items()`	和标准字典的items()方法有一点不同,该方法使用单值逻辑的**getitem**(): `>>> q = QueryDict('a=1&a=2&a=3')` `>>> q.items()` `[('a', '3')]`
`values()`	和标准字典的values()方法有一点不同,该方法使用单值逻辑的**getitem**():

此外, QueryDict也有一些方法，如下表：

方法	描述
`copy()`	返回对象的拷贝，内部实现是用Python标准库的copy.deepcopy()。该拷贝是mutable(可更改的) — 就是说，可以更改该拷贝的值。
`getlist(key)`	返回和参数key对应的所有值，作为一个Python list返回。如果key不存在，则返回空list。 It's guaranteed to return a list of some sort..
`setlist(key,list_)`	设置key的值为 `list_ (unlike __setitem__())`.
`appendlist(key,item)`	添加item到和key关联的内部list.
`setlistdefault(key,list)`	和setdefault有一点不同，它接受list而不是单个value作为参数。
`lists()`	和items()有一点不同, 它会返回key的所有值，作为一个list, 例如: `>>> q = QueryDict('a=1&a=2&a=3')` `>>> q.lists()` `[('a', ['1', '2', '3'])]`
`urlencode()`	返回一个以查询字符串格式进行格式化后的字符串(e.g., "a=2&b=3&b=5").

Django Admin 管理工具

Django 提供了基于 web 的管理工具。

Django自动管理工具是django.contrib的一部分。你可以在项目的 settings.py 中的 INSTALLED_APPS看到它：

```
INSTALLED_APPS = (
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
)
```

django.contrib是一套庞大的功能集，它是Django基本代码的组成部分。

激活管理工具

通常我们在生成项目时会在urls.py中自动设置好，我们只需去掉注释即可。

配置项如下所示：

```
from django.contrib import admin
admin.autodiscover()

# And include this URLpattern...
urlpatterns = patterns('',
    # ...
    (r'^admin/', include(admin.site.urls)),
    # ...
)
```

当这一切都配置好后，Django管理工具就可以运行了。

使用管理工具

启动开发服务器，然后在浏览器中访问：http://yoursite:8000/admin/，得到如下界面：

你可以通过命令 **python manage.py createsuperuser** 来创建超级用户，如下所示：

```
# python manage.py createsuperuser
Username (leave blank to use 'root'): admin
Email address: admin@w3cschool.cc
Password:
Password (again):
Superuser created successfully.
[root@solar HelloWorld]#
```

之后输入用户名密码登录，界面如下：

为了让admin界面管理某个数据模型，我们需要先注册该数据模型到admin。比如，我们之前在 TestModel 中已经创建了模型 Test 。修改 TestModel/admin.py:

```
from django.contrib import admin
from TestModel.models import Test

# Register your models here.
admin.site.register(Test)
```

刷新后即可看到 Testmodel 数据表:

复杂模型

管理页面的功能强大，完全有能力处理更加复杂的数据模型。

先在 TestModel/models.py 中增加一个更复杂的数据模型：

```
from django.db import models

# Create your models here.
class Contact(models.Model):
    name   = models.CharField(max_length=200)
    age    = models.IntegerField(default=0)
    email  = models.EmailField()
    def __unicode__(self):
        return self.name

class Tag(models.Model):
    contact = models.ForeignKey(Contact)
    name    = models.CharField(max_length=50)
    def __unicode__(self):
        return self.name
```

这里有两个表。Tag以Contact为外部键。一个Contact可以对应多个Tag。

我们还可以看到许多在之前没有见过的属性类型，比如IntegerField用于存储整数。

在 TestModel/admin.py 注册多个模型并显示：

```
from django.contrib import admin
from TestModel.models import Test,Contact,Tag

# Register your models here.
admin.site.register([Test, Contact, Tag])
```

刷新管理页面，显示结果如下：

在以上管理工具我们就能进行复杂模型操作。

自定义表单

我们可以自定义管理页面，来取代默认的页面。比如上面的"add"页面。我们想只显示name和email部分。修改 TestModel/admin.py:

```
from django.contrib import admin
from TestModel.models import Test,Contact,Tag

# Register your models here.
class ContactAdmin(admin.ModelAdmin):
    fields = ('name', 'email')

admin.site.register(Contact, ContactAdmin)
admin.site.register([Test, Tag])
```

以上代码定义了一个ContactAdmin类，用以说明管理页面的显示格式。

里面的fields属性定义了要显示的字段。

由于该类对应的是Contact数据模型，我们在注册的时候，需要将它们一起注册。显示效果如下：

我们还可以将输入栏分块，每个栏也可以定义自己的格式。修改TestModel/admin.py为：

```
from django.contrib import admin
from TestModel.models import Test,Contact,Tag

# Register your models here.
class ContactAdmin(admin.ModelAdmin):
    fieldsets = (
        ['Main',{
            'fields':('name','email'),
        }],
        ['Advance',{
            'classes': ('collapse',), # CSS
            'fields': ('age',),
        }]
    )

admin.site.register(Contact, ContactAdmin)
admin.site.register([Test, Tag])
```

上面的栏目分为了Main和Advance两部分。classes说明它所在的部分的CSS格式。这里让Advance部分隐藏：

Advance部分旁边有一个 Show 按钮，用于展开，展开后可点击 Hide 将其隐藏，如下图所示：

内联(Inline)显示

上面的Contact是Tag的外部键，所以有外部参考的关系。

而在默认的页面显示中，将两者分离开来，无法体现出两者的从属关系。我们可以使用内联显示，让Tag附加在Contact的编辑页面上显示。

修改TestModel/admin.py：

```
from django.contrib import admin
from TestModel.models import Test,Contact,Tag

# Register your models here.
class TagInline(admin.TabularInline):
    model = Tag

class ContactAdmin(admin.ModelAdmin):
    inlines = [TagInline]  # Inline
    fieldsets = (
        ['Main',{
            'fields':('name','email'),
        }],
        ['Advance',{
            'classes': ('collapse',),
            'fields': ('age',),
        }]

    )

admin.site.register(Contact, ContactAdmin)
admin.site.register([Test])
```

显示效果如下：

列表页的显示

在Contact输入数条记录后，Contact的列表页看起来如下:

我们也可以自定义该页面的显示，比如在列表中显示更多的栏目，只需要在ContactAdmin中增加list_display属性:

```
from django.contrib import admin
from TestModel.models import Test,Contact,Tag

# Register your models here.
class ContactAdmin(admin.ModelAdmin):
    list_display = ('name','age', 'email') # list

admin.site.register(Contact, ContactAdmin)
admin.site.register([Test, Tag])
```

刷新页面显示效果如下：

搜索功能在管理大量记录时非常有，我们可以使用search_fields为该列表页增加搜索栏：

```
from django.contrib import admin
from TestModel.models import Test,Contact,Tag

# Register your models here.
class ContactAdmin(admin.ModelAdmin):
    list_display = ('name','age', 'email')
    search_fields = ('name',)

admin.site.register(Contact, ContactAdmin)
admin.site.register([Test])
```

在本实例中我们搜索了 name 为 w3cschool.cc(本站域名) 的记录，显示结果如下：

Django Admin 管理工具还有非常多实用的功能，感兴趣的同学可以深入研究下。

Django Nginx+uwsgi 安装配置

在前面的章节中我们使用 **python manage.py runserver** 来运行服务器。这只适用测试环境中使用。

正式发布的服务，我们需要一个可以稳定而持续的服务器，比如apache, Nginx, lighttpd等，本文将以 Nginx 为例。

安装基础开发包

Centos 下安装步骤如下：

```
yum groupinstall "Development tools"
yum install zlib-devel bzip2-devel pcre-devel openssl-devel ncurses-devel sqlite-devel re
```

CentOS 自带 Python 2.4.3，但我们可以再安装Python2.7.5：

```
cd ~
wget http://python.org/ftp/python/2.7.5/Python-2.7.5.tar.bz2
tar xvf Python-2.7.5.tar.bz2
cd Python-2.7.5
./configure --prefix=/usr/local
make && make altinstall
```

安装Python包管理

easy_install包 https://pypi.python.org/pypi/distribute

安装步骤:

```
cd ~
wget https://pypi.python.org/packages/source/d/distribute/distribute-0.6.49.tar.gz
tar xf distribute-0.6.49.tar.gz
cd distribute-0.6.49
python2.7 setup.py install
easy_install --version
```

pip包: https://pypi.python.org/pypi/pip

安装pip的好处是可以 pip list、pip uninstall 管理Python包， easy_install 没有这个功能，只有uninstall

安装uwsgi

uwsgi:https://pypi.python.org/pypi/uWSGI

uwsgi参数详解：http://uwsgi-docs.readthedocs.org/en/latest/Options.html

```
pip install uwsgi
uwsgi --version    #查看 uwsgi 版本
```

测试uwsgi是否正常：

新建test.py文件，内容如下：

```
def application(env, start_response):
    start_response('200 OK', [('Content-Type','text/html')])
    return "Hello World"
```

然后在终端运行：

```
uwsgi --http :8001 --wsgi-file test.py
```

在浏览器内输入：http://127.0.0.1:8001，查看是否有"Hello World"输出，若没有输出，请检查你的安装过程。

安装 Django

```
pip install django
```

测试django是否正常，运行：

```
django-admin.py startproject demosite
cd demosite
python2.7 manage.py runserver 0.0.0.0:8002
```

在浏览器内输入：http://127.0.0.1:8002，检查django是否运行正常。

安装 Nginx

安装命令如下：

```
cd ~
wget http://nginx.org/download/nginx-1.5.6.tar.gz
tar xf nginx-1.5.6.tar.gz
cd nginx-1.5.6
./configure --prefix=/usr/local/nginx-1.5.6 \
--with-http_stub_status_module \
--with-http_gzip_static_module
make && make install
```

你可以阅读 Nginx 安装配置 了解更多内容。

uwsgi 配置

uwsgi支持ini、xml等多种配置方式，本文以 ini 为例， 在/ect/目录下新建uwsgi9090.ini，添加如下配置：

```
[uwsgi]
socket = 127.0.0.1:9090
master = true         //主进程
vhost = true          //多站模式
no-site = true        //多站模式时不设置入口模块和文件
workers = 2           //子进程数
reload-mercy = 10
vacuum = true         //退出、重启时清理文件
max-requests = 1000
limit-as = 512
buffer-size = 30000
pidfile = /var/run/uwsgi9090.pid    //pid文件，用于下面的脚本启动、停止该进程
daemonize = /website/uwsgi9090.log
```

Nginx 配置

找到nginx的安装目录（如：/usr/local/nginx/），打开conf/nginx.conf文件，修改server配置：

```
server {
        listen       80;
        server_name  localhost;

        location / {
            include  uwsgi_params;
            uwsgi_pass  127.0.0.1:9090;              //必须和uwsgi中的设置一致
            uwsgi_param UWSGI_SCRIPT demosite.wsgi;  //入口文件，即wsgi.py相对于项目根目录的位
            uwsgi_param UWSGI_CHDIR /demosite;       //项目根目录
            index  index.html index.htm;
            client_max_body_size 35m;
        }
    }
```

你可以阅读 Nginx 安装配置 了解更多内容。

设置完成后，在终端运行：

```
uwsgi --ini /etc/uwsgi9090.ini &
/usr/local/nginx/sbin/nginx
```

在浏览器输入：http://127.0.0.1，你就可以看到django的"It work"了。

W3School Node.js教程

来源：Node.js教程

整理：飞龙

Node.js 简介

简单的说 Node.js 就是运行在服务端的 JavaScript。

Node.js 是一个基于Chrome JavaScript 运行时建立的一个平台。

Node.js是一个事件驱动I/O服务端JavaScript环境，基于Google的V8引擎，V8引擎执行Javascript的速度非常快，性能非常好。

谁适合阅读本教程？

如果你是一个前端程序员，你不懂的像PHP、Python或Java等动态编程语言，然后你想创建自己的服务，那么Node.js是一个非常好的选择。

Node.js 是运行在服务端的 JavaScript，如果你熟悉Javascript，那么你将会很容易的学会Node.js。

当然，如果你是后端程序员，想部署一些高性能的服务，那么学习Node.js也是一个非常好的选择。

学习本教程前你需要了解

在继续本教程之前，你应该了解一些基本的计算机编程术语。如果你学习过Javascript,PHP，Java等编程语言，将有助于你更快的了解Node.js编程。

第一个**Node.js**程序：**Hello World**！

脚本模式

以下是我们的第一个Node.js程序：

```
console.log("Hello World");
```

保存该文件，文件名为 helloworld.js， 并通过 node命令来执行：

```
node helloworld.js
```

程序执行后，正常的话，就会在终端输出 Hello World。

交互模式

打开终端，键入node进入命令交互模式，可以输入一条代码语句后立即执行并显示结果，例如：

```
$ node
> console.log('Hello World!');
Hello World!
```

Node.js 安装配置

本章节我们将向大家介绍在window和Linux上安装Node.js的方法。

本安装教程以Node.js v0.10.26 版本为例。

Node.js安装包及源码下载地址为：http://www.nodejs.org/download/。

根据不同平台系统选择你需要的Node.js安装包。

注意：Linux上安装Node.js需要安装Python 2.6 或 2.7， 不建议安装Python 3.0以上版本。

Windowv 上安装Node.js

Windows 安装包(.msi)：

32 位安装包下载地址：http://nodejs.org/dist/v0.10.26/node-v0.10.26-x86.msi

64 位安装包下载地址：http://nodejs.org/dist/v0.10.26/x64/node-v0.10.26-x64.msi

安装步骤：

步骤 1 : 双击下载后的安装包 node-v0.10.26-x86.msi，如下所示：

步骤 2 : 点击以上的Run(运行)，将出现如下界面：

步骤 3 : 勾选接受协议选项，点击 next（下一步） 按钮 :

步骤 4 : Node.js默认安装目录为 "C:\Program Files\nodejs\" , 你可以修改目录，并点击 next（下一步）：

步骤 5 : 点击树形图标来选择你需要的安装模式 , 然后点击下一步 next（下一步）

步骤 6 :点击 Install（安装） 开始安装Node.js。你也可以点击 Back（返回）来修改先前的配置。 然后并点击 next（下一步）：

安装过程：

点击 Finish（完成）按钮退出安装向导。

检测PATH环境变量是否配置了Node.js，点击开始=》运行=》输入"cmd" => 输入命令"path"，输出如下结果：

```
PATH=C:\oraclexe\app\oracle\product\10.2.0\server\bin;C:\Windows\system32;
C:\Windows;C:\Windows\System32\Wbem;C:\Windows\System32\WindowsPowerShell\v1.0\;
c:\python32\python;C:\MinGW\bin;C:\Program Files\GTK2-Runtime\lib;
C:\Program Files\MySQL\MySQL Server 5.5\bin;C:\Program Files\nodejs\;
C:\Users\rg\AppData\Roaming\npm
```

我们可以看到环境变量中已经包含了C:\Program Files\nodejs\

检查Node.js版本

Windows 二进制文件 **(.exe)**安装：

32 位安装包下载地址：http://nodejs.org/dist/v0.10.26/node.exe

64 位安装包下载地址：http://nodejs.org/dist/v0.10.26/x64/node.exe

安装步骤

步骤 1：双击下载的安装包 Node.exe，将出现如下界面：

点击 Run（运行）按钮将出现命令行窗口：

版本测试

进入 node.exe 所在的目录，如下所示：

如果你获得以上输出结果，说明你已经成功安装了Node.js。

Linux上安装 Node.js

Ubuntu 安装

以下部分我们将介绍在Ubuntu Linux下安装 Node.js 。 其他的Linux系统，如Centos等类似如下安装步骤。

在 Github 上获取 Node.js 源码：

在完成下载后，将源码包名改为 'node'。

修改目录权限：

使用 './configure' 创建编译文件。

编译: make。

完成安装: make install。

最后我们输入'node --version' 命令来查看Node.js是否安装成功。

centOS下安装nodejs

1、下载源码，你需要在http://nodejs.org/下载最新的Nodejs版本，本文以v0.10.24为例:

```
cd /usr/local/src/
wget http://nodejs.org/dist/v0.10.24/node-v0.10.24.tar.gz
```

2、解压源码

```
tar zxvf node-v0.10.24.tar.gz
```

3、 编译安装

```
cd node-v0.10.24
./configure --prefix=/usr/local/node/0.10.24
make
make install
```

4、 配置NODE_HOME，进入profile编辑环境变量

```
vim /etc/profile
```

设置nodejs环境变量，在*export PATH USER LOGNAME MAIL HOSTNAME HISTSIZE HISTCONTROL* 一行的上面添加如下内容:

```
#set for nodejs
export NODE_HOME=/usr/local/node/0.10.24
export PATH=$NODE_HOME/bin:$PATH
```

:wq保存并退出，编译/etc/profile 使配置生效

```
source /etc/profile
```

验证是否安装配置成功

```
node -v
```

输出 v0.10.24 表示配置成功

npm模块安装路径

```
/usr/local/node/0.10.24/lib/node_modules/
```

注：Nodejs 官网提供了编译好的Linux二进制包，你也可以下载下来直接应用。

Node.js 创建第一个应用

如果我们使用PHP来编写后端的代码时，需要Apache 或者 Nginx 的HTTP 服务器，并配上 mod_php5 模块和php-cgi。

从这个角度看，整个"接收 HTTP 请求并提供 Web 页面"的需求根本不需 要 PHP 来处理。

不过对 Node.js 来说，概念完全不一样了。使用 Node.js 时，我们不仅仅 在实现一个应用，同时还实现了整个 HTTP 服务器。事实上，我们的 Web 应用以及对应的 Web 服务器基本上是一样的。

在我们创建 Node.js 第一个 "Hello, World!" 应用前，让我们先了解下 Node.js 应用是由哪几部分组成的：

1. 引入 **required** 模块：我们可以使用 **require** 指令来载入 Node.js 模块。
2. 创建服务器：服务器可以监听客户端的请求，类似于 Apache 、Nginx 等 HTTP 服务器。
3. 接收请求与响应请求 服务器很容易创建，客户端可以使用浏览器或终端发送 HTTP 请求，服务器接收请求后返回响应数据。

创建 Node.js 应用

步骤一、引入 required 模块

我们使用 **require** 指令来载入 http 模块，并将实例化的 HTTP 赋值给变量 http，实例如下:

```
var http =  require("http");
```

步骤一、创建服务器

接下来我们使用 http.createServer() 方法创建服务器，并使用 listen 方法绑定 8888 端口。 函数通过 request, response 参数来接收和响应数据。

实例如下，在你项目的根目录下创建一个叫 server.js 的文件，并写入以下代码：

```
var http =  require('http'); http.createServer(function  (request, response)  {  // 发送 H
```

以上代码我们完成了一个可以工作的 HTTP 服务器。

使用 **node** 命令执行以上的代码：

```
node server.js Server running at http://127.0.0.1:8888/
```

接下来，打开浏览器访问 http://127.0.0.1:8888/，你会看到一个写着 "Hello World"的网页。

分析**Node.js** 的 **HTTP** 服务器：

- 第一行请求（require）Node.js 自带的 http 模块，并且把它赋值给 http 变量。
- 接下来我们调用 http 模块提供的函数： createServer 。这个函数会返回 一个对象，这个对象有一个叫做 listen 的方法，这个方法有一个数值参数， 指定这个 HTTP 服务器监听的端口号。

Gif 实例演示

接下来我们通过 Gif 图为大家演示实例操作：

NPM 使用介绍

NPM是随同NodeJS一起安装的包管理工具，能解决NodeJS代码部署上的很多问题，常见的使用场景有以下几种：

- 允许用户从NPM服务器下载别人编写的第三方包到本地使用。
- 允许用户从NPM服务器下载并安装别人编写的命令行程序到本地使用。
- 允许用户将自己编写的包或命令行程序上传到NPM服务器供别人使用。

由于新版的nodejs已经集成了npm，所以之前npm也一并安装好了。同样可以通过输入 **"npm -v"** 来测试是否成功安装。命令如下，出现版本提示表示安装成功:

```
$ npm -v
2.3.0
```

如果你安装的是旧版本的 npm，可以很容易得通过 npm 命令来升级，命令如下：

```
$ sudo npm install npm -g
/usr/local/bin/npm -> /usr/local/lib/node_modules/npm/bin/npm-cli.js
npm@2.14.2 /usr/local/lib/node_modules/npm
```

使用 npm 命令安装模块

npm 安装 Node.js 模块语法格式如下：

```
$ npm install <Module Name>
```

以下实例，我们使用 npm 命令安装常用的 Node.js web框架模块 **express**:

```
$ npm install express
```

安装好之后，express 包就放在了工程目录下的 node_modules 目录中，因此在代码中只需要通过 **require('express')** 的方式就好，无需指定第三方包路径。

```
var express = require('express');
```

全局安装与本地安装

npm 的包安装分为本地安装（local）、全局安装（global）两种，从敲的命令行来看，差别只是有没有-g而已，比如

```
npm install express          # 本地安装
npm install express -g   # 全局安装
```

如果出现以下错误：

```
npm err! Error: connect ECONNREFUSED 127.0.0.1:8087
```

解决办法为：

```
$ npm config set proxy null
```

本地安装

- 1. 将安装包放在 ./node_modules 下（运行 npm 命令时所在的目录），如果没有 node_modules 目录，会在当前执行 npm 命令的目录下生成 node_modules 目录。
- 2. 可以通过 require() 来引入本地安装的包。

全局安装

- 1. 将安装包放在 /usr/local 下。
- 2. 可以直接在命令行里使用。
- 3. 不能通过 require() 来引入本地安装的包。

接下来我们使用全局方式安装 express

```
$ npm install express -g
```

安装过程输出如下内容，第一行输出了模块的版本号及安装位置。

```
express@4.13.3 node_modules/express
├── escape-html@1.0.2
├── range-parser@1.0.2
├── merge-descriptors@1.0.0
├── array-flatten@1.1.1
├── cookie@0.1.3
├── utils-merge@1.0.0
├── parseurl@1.3.0
├── cookie-signature@1.0.6
├── methods@1.1.1
├── fresh@0.3.0
├── vary@1.0.1
├── path-to-regexp@0.1.7
├── content-type@1.0.1
├── etag@1.7.0
├── serve-static@1.10.0
├── content-disposition@0.5.0
├── depd@1.0.1
├── qs@4.0.0
├── finalhandler@0.4.0 (unpipe@1.0.0)
├── on-finished@2.3.0 (ee-first@1.1.1)
├── proxy-addr@1.0.8 (forwarded@0.1.0, ipaddr.js@1.0.1)
├── debug@2.2.0 (ms@0.7.1)
├── type-is@1.6.8 (media-typer@0.3.0, mime-types@2.1.6)
├── accepts@1.2.12 (negotiator@0.5.3, mime-types@2.1.6)
└── send@0.13.0 (destroy@1.0.3, statuses@1.2.1, ms@0.7.1, mime@1.3.4, http-errors@1.3.1)
```

你可以使用以下命令来查看所有全局安装的模块：

```
$ npm ls -g
```

使用 package.json

package.json 位于模块的目录下，用于定义包的属性。接下来让我们来看下 express 包的 package.json 文件，位于 node_modules/express/package.json 内容：

```
{
  "name": "express",
  "description": "Fast, unopinionated, minimalist web framework",
  "version": "4.13.3",
  "author": {
    "name": "TJ Holowaychuk",
    "email": "tj@vision-media.ca"
  },
  "contributors": [
    {
      "name": "Aaron Heckmann",
      "email": "aaron.heckmann+github@gmail.com"
    },
    {
      "name": "Ciaran Jessup",
      "email": "ciaranj@gmail.com"
    },
    {
      "name": "Douglas Christopher Wilson",
      "email": "doug@somethingdoug.com"
    },
    {
      "name": "Guillermo Rauch",
      "email": "rauchg@gmail.com"
```

```
    },
    {
      "name": "Jonathan Ong",
      "email": "me@jongleberry.com"
    },
    {
      "name": "Roman Shtylman",
      "email": "shtylman+expressjs@gmail.com"
    },
    {
      "name": "Young Jae Sim",
      "email": "hanul@hanul.me"
    }
  ],
  "license": "MIT",
  "repository": {
    "type": "git",
    "url": "git+https://github.com/strongloop/express.git"
  },
  "homepage": "http://expressjs.com/",
  "keywords": [
    "express",
    "framework",
    "sinatra",
    "web",
    "rest",
    "restful",
    "router",
    "app",
    "api"
  ],
  "dependencies": {
    "accepts": "~1.2.12",
    "array-flatten": "1.1.1",
    "content-disposition": "0.5.0",
    "content-type": "~1.0.1",
    "cookie": "0.1.3",
    "cookie-signature": "1.0.6",
    "debug": "~2.2.0",
    "depd": "~1.0.1",
    "escape-html": "1.0.2",
    "etag": "~1.7.0",
    "finalhandler": "0.4.0",
    "fresh": "0.3.0",
    "merge-descriptors": "1.0.0",
    "methods": "~1.1.1",
    "on-finished": "~2.3.0",
    "parseurl": "~1.3.0",
    "path-to-regexp": "0.1.7",
    "proxy-addr": "~1.0.8",
    "qs": "4.0.0",
    "range-parser": "~1.0.2",
    "send": "0.13.0",
    "serve-static": "~1.10.0",
    "type-is": "~1.6.6",
    "utils-merge": "1.0.0",
    "vary": "~1.0.1"
  },
  "devDependencies": {
    "after": "0.8.1",
    "ejs": "2.3.3",
    "istanbul": "0.3.17",
    "marked": "0.3.5",
    "mocha": "2.2.5",
    "should": "7.0.2",
    "supertest": "1.0.1",
    "body-parser": "~1.13.3",
    "connect-redis": "~2.4.1",
    "cookie-parser": "~1.3.5",
    "cookie-session": "~1.2.0",
    "express-session": "~1.11.3",
    "jade": "~1.11.0",
```

```
    "method-override": "~2.3.5",
    "morgan": "~1.6.1",
    "multiparty": "~4.1.2",
    "vhost": "~3.0.1"
  },
  "engines": {
    "node": ">= 0.10.0"
  },
  "files": [
    "LICENSE",
    "History.md",
    "Readme.md",
    "index.js",
    "lib/"
  ],
  "scripts": {
    "test": "mocha --require test/support/env --reporter spec --bail --check-leaks test/
    "test-ci": "istanbul cover node_modules/mocha/bin/_mocha --report lcovonly -- --requi
    "test-cov": "istanbul cover node_modules/mocha/bin/_mocha -- --require test/support/e
    "test-tap": "mocha --require test/support/env --reporter tap --check-leaks test/ test
  },
  "gitHead": "ef7ad681b245fba023843ce94f6bcb8e275bbb8e",
  "bugs": {
    "url": "https://github.com/strongloop/express/issues"
  },
  "_id": "express@4.13.3",
  "_shasum": "ddb2f1fb4502bf33598d2b032b037960ca6c80a3",
  "_from": "express@*",
  "_npmVersion": "1.4.28",
  "_npmUser": {
    "name": "dougwilson",
    "email": "doug@somethingdoug.com"
  },
  "maintainers": [
    {
      "name": "tjholowaychuk",
      "email": "tj@vision-media.ca"
    },
    {
      "name": "jongleberry",
      "email": "jonathanrichardong@gmail.com"
    },
    {
      "name": "dougwilson",
      "email": "doug@somethingdoug.com"
    },
    {
      "name": "rfeng",
      "email": "enjoyjava@gmail.com"
    },
    {
      "name": "aredridel",
      "email": "aredridel@dinhe.net"
    },
    {
      "name": "strongloop",
      "email": "callback@strongloop.com"
    },
    {
      "name": "defunctzombie",
      "email": "shtylman@gmail.com"
    }
  ],
  "dist": {
    "shasum": "ddb2f1fb4502bf33598d2b032b037960ca6c80a3",
    "tarball": "http://registry.npmjs.org/express/-/express-4.13.3.tgz"
  },
  "directories": {},
  "_resolved": "https://registry.npmjs.org/express/-/express-4.13.3.tgz",
  "readme": "ERROR: No README data found!"
}
```

Package.json 属性说明

- **name** - 包名。
- **version** - 包的版本号。
- **description** - 包的描述。
- **homepage** - 包的官网 url 。
- **author** - 包的作者姓名。
- **contributors** - 包的其他贡献者姓名。
- **dependencies** - 依赖包列表。如果依赖包没有安装，npm 会自动将依赖包安装在 node_module 目录下。
- **repository** - 包代码存放的地方的类型，可以是 git 或 svn，git 可在 Github 上。
- **main** - main 字段是一个模块ID，它是一个指向你程序的主要项目。就是说，如果你包的名字叫 express，然后用户安装它，然后require("express")。
- **keywords** - 关键字

卸载模块

我们可以使用以下命令来卸载 Node.js 模块。

```
$ npm uninstall express
```

卸载后，你可以到 /node_modules/ 目录下查看包是否还存在，或者使用以下命令查看：

```
$ npm ls
```

更新模块

我们可以使用以下命令更新模块：

```
$ npm update express
```

搜索模块

使用以下来搜索模块：

```
$ npm search express
```

创建模块

创建模块，package.json 文件是必不可少的。我们可以使用 NPM 生成 package.json 文件，生成的文件包含了基本的结果。

```
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg> --save` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
name: (node_modules) runoob                   # 模块名
version: (1.0.0)
description: Node.js 测试模块(www.runoob.com)  # 描述
entry point: (index.js)
test command: make test
git repository: https://github.com/runoob/runoob.git  # Github 地址
keywords:
author:
license: (ISC)
About to write to ……/node_modules/package.json:      # 生成地址

{
  "name": "runoob",
  "version": "1.0.0",
  "description": "Node.js 测试模块(www.runoob.com)",
  ……
}

Is this ok? (yes) yes
```

以上的信息，你需要根据你自己的情况输入。在最后输入 "yes" 后会生成 package.json 文件。

接下来我们可以使用以下命令在 npm 资源库中注册用户（使用邮箱注册）：

```
$ npm adduser
Username: mcmohd
Password:
Email: (this IS public) mcmohd@gmail.com
```

接下来我们就用以下命令来发布模块：

```
$ npm publish
```

如果你以上的步骤都操作正确，你就可以跟其他模块一样使用 npm 来安装。

版本号

使用NPM下载和发布代码时都会接触到版本号。NPM使用语义版本号来管理代码，这里简单介绍一下。

语义版本号分为X.Y.Z三位，分别代表主版本号、次版本号和补丁版本号。当代码变更时，版本号按以下原则更新。

- 如果只是修复bug，需要更新Z位。
- 如果是新增了功能，但是向下兼容，需要更新Y位。
- 如果有大变动，向下不兼容，需要更新X位。

版本号有了这个保证后，在申明第三方包依赖时，除了可依赖于一个固定版本号外，还可依赖于某个范围的版本号。例如"argv": "0.0.x"表示依赖于0.0.x系列的最新版argv。

NPM支持的所有版本号范围指定方式可以查看官方文档。

NPM 常用命令

除了本章介绍的部分外，NPM还提供了很多功能，package.json里也有很多其它有用的字段。

除了可以在npmjs.org/doc/查看官方文档外，这里再介绍一些NPM常用命令。

NPM提供了很多命令，例如install和publish，使用npm help可查看所有命令。

- NPM提供了很多命令，例如 `install` 和 `publish` ，使用 `npm help` 可查看所有命令。
- 使用 `npm help <command>` 可查看某条命令的详细帮助，例如 `npm help install` 。
- 在 `package.json` 所在目录下使用 `npm install . -g` 可先在本地安装当前命令行程序，可用于发布前的本地测试。
- 使用 `npm update <package>` 可以把当前目录下 `node_modules` 子目录里边的对应模块更新至最新版本。
- 使用 `npm update <package> -g` 可以把全局安装的对应命令行程序更新至最新版。
- 使用 `npm cache clear` 可以清空NPM本地缓存，用于对付使用相同版本号发布新版本代码的人。
- 使用 `npm unpublish <package>@<version>` 可以撤销发布自己发布过的某个版本代码。

Node.js REPL(交互式解释器)

Node.js REPL(Read Eval Print Loop:交互式解释器) 表示一个电脑的环境，类似 Window 系统的终端或 Unix/Linux shell，我们可以在终端中输入命令，并接收系统的响应。

Node 自带了交互式解释器，可以执行以下任务：

- 读取 - 读取用户输入，解析输入了Javascript 数据结构并存储在内存中。
- 执行 - 执行输入的数据结构
- 打印 - 输出结果
- 循环 - 循环操作以上步骤直到用户两次按下 **ctrl-c** 按钮退出。

Node 的交互式解释器可以很好的调试 Javascript 代码。

开始学习 REPL

我们可以输入以下命令来启动 Node 的终端：

```
$ node
>
```

这时我们就可以在 > 后输入简单的表达式，并按下回车键来计算结果。

简单的表达式运算

接下来让我们在 Node.js REPL 的命令行窗口中执行简单的数学运算：

```
$ node
> 1 +4
5
> 5 / 2
2.5
> 3 * 6
18
> 4 - 1
3
> 1 + ( 2 * 3 ) - 4
3
>
```

使用变量

你可以将数据存储在变量中，并在你需要的使用它。

变量声明需要使用 **var** 关键字，如果没有使用 var 关键字变量会直接打印出来。

使用 **var** 关键字的变量可以使用 console.log() 来输出变量。

```
$ node
> x = 10
10
> var y = 10
undefined
> x + y
20
> console.log("Hello World")
Hello World
undefined
> console.log("www.runoob.com")
www.runoob.com
undefined
```

多行表达式

Node REPL 支持输入多行表达式，这就有点类似 JavaScript。接下来让我们来执行一个 do-while 循环：

```
$ node
> var x = 0
undefined
> do {
... x++;
... console.log("x: " + x);
... } while ( x < 5 );
x: 1
x: 2
x: 3
x: 4
x: 5
undefined
>
```

... 三个点的符号是系统自动生成的，你回车换行后即可。Node 会自动检测是否为连续的表达式。

下划线(_)变量

你可以使用下划线(_)获取表达式的运算结果：

```
$ node
> var x = 10
undefined
> var y = 20
undefined
> x + y
30
> var sum = _
undefined
> console.log(sum)
30
undefined
>
```

REPL 命令

- **ctrl + c** - 退出当前终端。
- **ctrl + c 按下两次** - 退出 Node REPL。
- **ctrl + d** - 退出 Node REPL.
- **向上/向下 键** - 查看输入的历史命令
- **tab 键** - 列出当前命令
- **.help** - 列出使用命令
- **.break** - 退出多行表达式
- **.clear** - 退出多行表达式
- **.save *filename*** - 保存当前的 Node REPL 会话到指定文件
- **.load *filename*** - 载入当前 Node REPL 会话的文件内容。

停止 REPL

前面我们已经提到按下两次 **ctrl + c** 建就能退出 REPL:

```
$ node
>
(^C again to quit)
>
```

Gif 实例演示

接下来我们通过 Gif 图为大家演示实例操作：

Node.js 回调函数

Node.js 异步编程的直接体现就是回调。

异步编程依托于回调来实现，但不能说使用了回调后程序就异步化了。

回调函数在完成任务后就会被调用，Node 使用了大量的回调函数，Node 所有 API 都支持回调函数。

例如，我们可以一边读取文件，一边执行其他命令，在文件读取完成后，我们将文件内容作为回调函数的参数返回。这样在执行代码时就没有阻塞或等待文件 I/O 操作。这就大大提高了 Node.js 的性能，可以处理大量的并发请求。

阻塞代码实例

创建一个文件 input.txt ，内容如下：

```
菜鸟教程官网地址：www.runoob.com
```

创建 main.js 文件, 代码如下：

```
var fs = require("fs");

var data = fs.readFileSync('input.txt');

console.log(data.toString());
console.log("程序执行结束!");
```

以上代码执行结果如下：

```
$ node main.js
菜鸟教程官网地址：www.runoob.com

程序执行结束!
```

非阻塞代码实例

创建一个文件 input.txt ，内容如下：

```
菜鸟教程官网地址：www.runoob.com
```

创建 main.js 文件, 代码如下：

```
var fs = require("fs");

fs.readFile('input.txt', function (err, data) {
    if (err) return console.error(err);
    console.log(data.toString());
});

console.log("程序执行结束!");
```

以上代码执行结果如下：

```
$ node main.js
程序执行结束!
菜鸟教程官网地址：www.runoob.com
```

以上两个实例我们了解了阻塞与非阻塞调用的不同。第一个实例在文件读取完后才执行完程序。 第二个实例我们呢不需要等待文件读取完，这样就可以在读取文件时同时执行接下来的代码，大大提高了程序的性能。

因此，阻塞按是按顺序执行的，而非阻塞是不需要按顺序的，所以如果需要处理回调函数的参数，我们就需要写在回调函数内。

Node.js 事件循环

Node.js 是单进程单线程应用程序，但是通过事件和回调支持并发，所以性能非常高。

Node.js 的每一个 API 都是异步的，并作为一个独立线程运行，使用异步函数调用，并处理并发。

Node.js 基本上所有的事件机制都是用设计模式中观察者模式实现。

Node.js 单线程类似进入一个while(true)的事件循环，直到没有事件观察者退出，每个异步事件都生成一个事件观察者，如果有事件发生就调用该回调函数.

事件驱动程序

Node.js 使用事件驱动模型，当web server接收到请求，就把它关闭然后进行处理，然后去服务下一个web请求。

当这个请求完成，它被放回处理队列，当到达队列开头，这个结果被返回给用户。

这个模型非常高效可扩展性非常强，因为webserver一直接受请求而不等待任何读写操作。（这也被称之为非阻塞式IO或者事件驱动IO）

在事件驱动模型中，会生成一个主循环来监听事件，当检测到事件时触发回调函数。

整个事件驱动的流程就是这么实现的，非常简洁。有点类似于观察者模式，事件相当于一个主题(Subject)，而所有注册到这个事件上的处理函数相当于观察者(Observer)。

Node.js 有多个内置的事件，我们可以通过引入 events 模块，并通过实例化 EventEmitter 类来绑定和监听事件，如下实例：

```
// 引入 events 模块
var events = require('events');
// 创建 eventEmitter 对象
var eventEmitter = new events.EventEmitter();
```

以下程序绑定事件处理程序：

```
// 绑定事件及事件的处理程序
eventEmitter.on('eventName', eventHandler);
```

我们可以通过程序触发事件：

```
// 触发事件
eventEmitter.emit('eventName');
```

实例

创建 main.js 文件，代码如下所示：

```
// 引入 events 模块
var events = require('events');
// 创建 eventEmitter 对象
var eventEmitter = new events.EventEmitter();

// 创建事件处理程序
var connectHandler = function connected() {
   console.log('连接成功。');

   // 触发 data_received 事件
   eventEmitter.emit('data_received');
}

// 绑定 connection 事件处理程序
eventEmitter.on('connection', connectHandler);

// 使用匿名函数绑定 data_received 事件
eventEmitter.on('data_received', function(){
   console.log('数据接收成功。');
});

// 触发 connection 事件
eventEmitter.emit('connection');

console.log("程序执行完毕。");
```

接下来让我们执行以上代码：

```
$ node main.js
连接成功。
数据接收成功。
程序执行完毕。
```

Node 应用程序是如何工作的？

在 Node 应用程序中，执行异步操作的函数将回调函数作为最后一个参数， 回调函数接收错误对象作为第一个参数。

接下来让我们来重新看下前面的实例，创建一个 input.txt ,文件内容如下：

```
菜鸟教程官网地址：www.runoob.com
```

创建 main.js 文件，代码如下：

```
var fs = require("fs");

fs.readFile('input.txt', function (err, data) {
   if (err){
      console.log(err.stack);
      return;
   }
   console.log(data.toString());
});
console.log("程序执行完毕");
```

以上程序中 fs.readFile() 是异步函数用于读取文件。 如果在读取文件过程中发生错误，错误 err 对象就会输出错误信息。

如果没发生错误，readFile 跳过 err 对象的输出，文件内容就通过回调函数输出。

执行以上代码，执行结果如下：

```
程序执行完毕
菜鸟教程官网地址：www.runoob.com
```

接下来我们删除 input.txt 文件，执行结果如下所示：

```
程序执行完毕
Error: ENOENT, open 'input.txt'
```

因为文件 input.txt 不存在，所以输出了错误信息。

Node.js EventEmitter

Node.js 所有的异步 I/O 操作在完成时都会发送一个事件到事件队列。

Node.js里面的许多对象都会分发事件：一个net.Server对象会在每次有新连接时分发一个事件， 一个fs.readStream对象会在文件被打开的时候发出一个事件。 所有这些产生事件的对象都是 events.EventEmitter 的实例。

EventEmitter 类

events 模块只提供了一个对象： events.EventEmitter。EventEmitter 的核心就是事件触发与事件监听器功能的封装。

你可以通过require("events");来访问该模块。

```
// 引入 events 模块
var events = require('events');
// 创建 eventEmitter 对象
var eventEmitter = new events.EventEmitter();
```

EventEmitter 对象如果在实例化时发生错误，会触发 'error' 事件。当添加新的监听器时，'newListener' 事件会触发，当监听器被移除时，'removeListener' 事件被触发。

下面我们用一个简单的例子说明 EventEmitter 的用法：

```
//event.js 文件
var EventEmitter = require('events').EventEmitter;
var event = new EventEmitter();
event.on('some_event', function() {
    console.log('some_event 事件触发');
});
setTimeout(function() {
    event.emit('some_event');
}, 1000);
```

执行结果如下：

运行这段代码，1 秒后控制台输出了 **'some_event 事件触发'**。其原理是 event 对象注册了事件 some_event 的一个监听器，然后我们通过 setTimeout 在 1000 毫秒以后向 event 对象发送事件 some_event，此时会调用some_event 的监听器。

```
$ node event.js
some_event 事件触发
```

EventEmitter 的每个事件由一个事件名和若干个参数组成，事件名是一个字符串，通常表达一定的语义。对于每个事件，EventEmitter 支持 若干个事件监听器。

当事件触发时，注册到这个事件的事件监听器被依次调用，事件参数作为回调函数参数传递。

让我们以下面的例子解释这个过程：

```
//event.js 文件
var events = require('events');
var emitter = new events.EventEmitter();
emitter.on('someEvent', function(arg1, arg2) {
    console.log('listener1', arg1, arg2);
});
emitter.on('someEvent', function(arg1, arg2) {
    console.log('listener2', arg1, arg2);
});
emitter.emit('someEvent', 'arg1 参数', 'arg2 参数');
```

执行以上代码，运行的结果如下：/p>

```
$ node event.js
listener1 arg1 参数 arg2 参数
listener2 arg1 参数 arg2 参数
```

以上例子中，emitter 为事件 someEvent 注册了两个事件监听器，然后触发了 someEvent 事件。

运行结果中可以看到两个事件监听器回调函数被先后调用。 这就是EventEmitter最简单的用法。

EventEmitter 提供了多个属性，如 **on** 和 **emit**。**on** 函数用于绑定事件函数，**emit** 属性用于触发一个事件。接下来我们来具体看下 EventEmitter 的属性介绍。

方法

方法	
addListener(event, listener)	为指定事件添加一个监听器到监听器数组的尾部。
on(event, listener)	为指定事件注册一个监听器，接受一个字符串 event 和一
once(event, listener)	为指定事件注册一个单次监听器，即 监听器最多只会触发 `server.once('connection', function (stream) { console.l`
removeListener(event, listener)	移除指定事件的某个监听器， 监听器 必须是该事件已经注 `var callback = function(stream) { console.log('someone`
removeAllListeners([event])	移除所有事件的所有监听器， 如果指定事件，则移除指定
setMaxListeners(n)	默认情况下， EventEmitters 如果你添加的监听器超过 10
listeners(event)	返回指定事件的监听器数组。
emit(event, [arg1], [arg2], [...])	按参数的顺序执行每个监听器，如果事件有注册监听返回

类方法

方法	描述
listenerCount(emitter, event)	返回指定事件的监听器数量。

事件

事件	描述
newListener	**event** - 字符串，事件名称 **listener** - 处理事件函数 该事件在添加新监听器时被触发。
removeListener	**event** - 字符串，事件名称 **listener** - 处理事件函数 从指定监听器数组中删除一个监听器。需要注意的是，此操作将会改变处于被删监听器之后的那些监听器的索引。

实例

以下实例通过 connection（连接）事件演示了 EventEmitter 类的应用。

创建 main.js 文件，代码如下：

```
var events = require('events');
var eventEmitter = new events.EventEmitter();

// 监听器 #1
var listner1 = function listner1() {
   console.log('监听器 listner1 执行。');
}

// 监听器 #2
var listner2 = function listner2() {
  console.log('监听器 listner2 执行。');
}

// 绑定 connection 事件，处理函数为 listner1
eventEmitter.addListener('connection', listner1);

// 绑定 connection 事件，处理函数为 listner2
eventEmitter.on('connection', listner2);

var eventListeners = require('events').EventEmitter.listenerCount(eventEmitter,'connectio
console.log(eventListeners + " 监听器监听连接事件。");

// 处理 connection 事件
eventEmitter.emit('connection');

// 移除监绑定的 listner1 函数
eventEmitter.removeListener('connection', listner1);
console.log("listner1 不再受监听。");

// 触发连接事件
eventEmitter.emit('connection');

eventListeners = require('events').EventEmitter.listenerCount(eventEmitter,'connection');
console.log(eventListeners + " 监听器监听连接事件。");

console.log("程序执行完毕。");
```

以上代码，执行结果如下所示：

```
$ node main.js
2 监听器监听连接事件。
监听器 listner1 执行。
监听器 listner2 执行。
listner1 不再受监听。
监听器 listner2 执行。
1 监听器监听连接事件。
程序执行完毕。
```

error 事件

EventEmitter 定义了一个特殊的事件 error，它包含了错误的语义，我们在遇到 异常的时候通常会触发 error 事件。

当 error 被触发时，EventEmitter 规定如果没有响 应的监听器，Node.js 会把它当作异常，退出程序并输出错误信息。

我们一般要为会触发 error 事件的对象设置监听器，避免遇到错误后整个程序崩溃。例如：

```
var events = require('events');
var emitter = new events.EventEmitter();
emitter.emit('error');
```

运行时会显示以下错误：

```
node.js:201
throw e; // process.nextTick error, or 'error' event on first tick
^
Error: Uncaught, unspecified 'error' event.
at EventEmitter.emit (events.js:50:15)
at Object.<anonymous> (/home/byvoid/error.js:5:9)
at Module._compile (module.js:441:26)
at Object..js (module.js:459:10)
at Module.load (module.js:348:31)
at Function._load (module.js:308:12)
at Array.0 (module.js:479:10)
at EventEmitter._tickCallback (node.js:192:40)
```

继承 EventEmitter

大多数时候我们不会直接使用 EventEmitter，而是在对象中继承它。包括 fs、net、 http 在内的，只要是支持事件响应的核心模块都是 EventEmitter 的子类。

为什么要这样做呢？原因有两点：

首先，具有某个实体功能的对象实现事件符合语义， 事件的监听和发射应该是一个对象的方法。

其次 JavaScript 的对象机制是基于原型的，支持 部分多重继承，继承 EventEmitter 不会打乱对象原有的继承关系。

Node.js Buffer(缓冲区)

JavaScript 语言自身只有字符串数据类型，没有二进制数据类型。

但在处理像TCP流或文件流时，必须使用到二进制数据。因此在 Node.js中，定义了一个 Buffer 类，该类用来创建一个专门存放二进制数据的缓存区。

在 Node.js 中，Buffer 类是随 Node 内核一起发布的核心库。Buffer 库为 Node.js 带来了一种存储原始数据的方法，可以让 Node.js 处理二进制数据，每当需要在 Node.js 中处理I/O操作中移动的数据时，就有可能使用 Buffer 库。原始数据存储在 Buffer 类的实例中。一个 Buffer 类似于一个整数数组，但它对应于 V8 堆内存之外的一块原始内存。

创建 Buffer 类

Node Buffer 类可以通过多种方式来创建。

方法 1

创建长度为 10 字节的 Buffer 实例：

```
var buf = new Buffer(10);
```

方法 2

通过给定的数组创建 Buffer 实例：

```
var buf = new Buffer([10, 20, 30, 40, 50]);
```

方法 3

通过一个字符串来创建 Buffer 实例：

```
var buf = new Buffer("www.runoob.com", "utf-8");
```

utf-8 是默认的编码方式，此外它同样支持以下编码："ascii", "utf8", "utf16le", "ucs2", "base64" 和 "hex"。

写入缓冲区

语法

写入 Node 缓冲区的语法如下所示：

```
buf.write(string[, offset][, length][, encoding])
```

参数

参数描述如下：

- **string** - 写入缓冲区的字符串。
- **offset** - 缓冲区开始写入的索引值，默认为 0 。
- **length** - 写入的字节数，默认为 buffer.length
- **encoding** - 使用的编码。默认为 'utf8' 。

返回值

返回实际写入的大小。如果 buffer 空间不足， 则只会写入部分字符串。

实例

```
buf = new Buffer(256);
len = buf.write("www.runoob.com");

console.log("写入字节数 : "+  len);
```

执行以上代码，输出结果为：

```
$node main.js
写入字节数 : 14
```

从缓冲区读取数据

语法

读取 Node 缓冲区数据的语法如下所示：

```
buf.toString([encoding][, start][, end])
```

参数

参数描述如下：

- **encoding** - 使用的编码。默认为 'utf8' 。
- **start** - 指定开始读取的索引位置，默认为 0。
- **end** - 结束位置，默认为缓冲区的末尾。

返回值

解码缓冲区数据并使用指定的编码返回字符串。

实例

```
buf = new Buffer(26);
for (var i = 0 ; i < 26 ; i++) {
  buf[i] = i + 97;
}

console.log( buf.toString('ascii'));       // 输出: abcdefghijklmnopqrstuvwxyz
console.log( buf.toString('ascii',0,5));   // 输出: abcde
console.log( buf.toString('utf8',0,5));    // 输出: abcde
console.log( buf.toString(undefined,0,5)); // 使用 'utf8' 编码, 并输出: abcde
```

执行以上代码，输出结果为：

```
$ node main.js
abcdefghijklmnopqrstuvwxyz
abcde
abcde
abcde
```

将 Buffer 转换为 JSON 对象

语法

将 Node Buffer 转换为 JSON 对象的函数语法格式如下：

```
buf.toJSON()
```

返回值

返回 JSON 对象。

实例

```
var buf = new Buffer('www.runoob.com');
var json = buf.toJSON(buf);

console.log(json);
```

执行以上代码，输出结果为：

```
[ 119, 119, 119, 46, 114, 117, 110, 111, 111, 98, 46, 99, 111, 109 ]
```

缓冲区合并

语法

Node 缓冲区合并的语法如下所示：

```
Buffer.concat(list[, totalLength])
```

参数

参数描述如下：

- **list** - 用于合并的 Buffer 对象数组列表。
- **totalLength** - 指定合并后Buffer对象的总长度。

返回值

返回一个多个成员合并的新 Buffer 对象。

实例

```
var buffer1 = new Buffer('菜鸟教程 ');
var buffer2 = new Buffer('www.runoob.com');
var buffer3 = Buffer.concat([buffer1,buffer2]);
console.log("buffer3 内容: " + buffer3.toString());
```

执行以上代码，输出结果为：

```
buffer3 内容: 菜鸟教程 www.runoob.com
```

缓冲区比较

语法

Node Buffer 比较的函数语法如下所示：

```
buf.compare(otherBuffer);
```

参数

参数描述如下：

- **otherBuffer** - 与 **buf** 对象比较的另外一个 Buffer 对象。

返回值

返回一个数字，表示 **buf** 在 **otherBuffer** 之前，之后或相同。

实例

```
var buffer1 = new Buffer('ABC');
var buffer2 = new Buffer('ABCD');
var result = buffer1.compare(buffer2);

if(result < 0) {
   console.log(buffer1 + " 在 " + buffer2 + "之前");
}else if(result == 0){
   console.log(buffer1 + " 与 " + buffer2 + "相同");
}else {
   console.log(buffer1 + " 在 " + buffer2 + "之后");
}
```

执行以上代码，输出结果为：

```
ABC在ABCD之前
```

拷贝缓冲区

语法

Node 缓冲区拷贝语法如下所示：

```
buf.copy(targetBuffer[, targetStart][, sourceStart][, sourceEnd])
```

参数

参数描述如下：

- **targetBuffer** - 要拷贝的 Buffer 对象。
- **targetStart** - 数字, 可选, 默认: 0
- **sourceStart** - 数字, 可选, 默认: 0
- **sourceEnd** - 数字, 可选, 默认: buffer.length

返回值

没有返回值。

实例

```
var buffer1 = new Buffer('ABC');
// 拷贝一个缓冲区
var buffer2 = new Buffer(3);
buffer1.copy(buffer2);
console.log("buffer2 content: " + buffer2.toString());
```

执行以上代码，输出结果为：

```
buffer2 content: ABC
```

缓冲区裁剪

Node 缓冲区裁剪语法如下所示：

```
buf.slice([start][, end])
```

参数

参数描述如下：

- **start** - 数字, 可选, 默认: 0
- **end** - 数字, 可选, 默认: buffer.length

返回值

返回一个新的缓冲区，它和旧缓冲区指向同一块内存，但是从索引 start 到 end 的位置剪切。

实例

```
var buffer1 = new Buffer('runoob');
// 剪切缓冲区
var buffer2 = buffer1.slice(0,2);
console.log("buffer2 content: " + buffer2.toString());
```

执行以上代码，输出结果为：

```
buffer2 content: ru
```

缓冲区长度

语法

Node 缓冲区长度计算语法如下所示：

```
buf.length;
```

返回值

返回 Buffer 对象所占据的内存长度。

实例

```
var buffer = new Buffer('www.runoob.com');
//  缓冲区长度
console.log("buffer length: " + buffer.length);
```

执行以上代码，输出结果为：

```
buffer length: 14
```

方法参考手册

以下列出了 Node.js Buffer 模块常用的方法（注意有些方法在旧版本是没有的）：

方法	描述
new Buffer(size)	分配一个新的 size 大小单位为8位字节的 buffer。 注意, size 将会抛出异常 RangeError。
new Buffer(buffer)	拷贝参数 buffer 的数据到 Buffer 实例。

new Buffer(str[, encoding])	分配一个新的 buffer，其中包含着传入的 str 字符串。 enco
buf.length	返回这个 buffer 的 bytes 数。注意这未必是 buffer 里面内容 分配的内存数，它不会随着这个 buffer 对象内容的改变而改
buf.write(string[, offset][, length][, encoding])	根据参数 offset 偏移量和指定的 encoding 编码方式，将参数 偏移量默认值是 0, encoding 编码方式默认是 utf8。 length 大小。 返回 number 类型，表示写入了多少 8 位字节流。如 个 string，它将只会只写入部分字符串。 length 默认是 buff 出现写入部分字符。
buf.writeUIntLE(value, offset, byteLength[, noAssert])	将value 写入到 buffer 里， 它由offset 和 byteLength 决定， `var b = new Buffer(6); b.writeUIntBE(0x1234567890ab, 0, 6` noAssert 值为 true 时，不再验证 value 和 offset 的有效性。
buf.writeUIntBE(value, offset, byteLength[, noAssert])	将value 写入到 buffer 里， 它由offset 和 byteLength 决定， true 时，不再验证 value 和 offset 的有效性。 默认是 false。
buf.writeIntLE(value, offset, byteLength[, noAssert])	将value 写入到 buffer 里， 它由offset 和 byteLength 决定， true 时，不再验证 value 和 offset 的有效性。 默认是 false。
buf.writeIntBE(value, offset, byteLength[, noAssert])	将value 写入到 buffer 里， 它由offset 和 byteLength 决定， true 时，不再验证 value 和 offset 的有效性。 默认是 false。
buf.readUIntLE(offset, byteLength[, noAssert])	支持读取 48 位以下的数字。noAssert 值为 true 时， offset 度，默认为 false。
buf.readUIntBE(offset, byteLength[, noAssert])	支持读取 48 位以下的数字。noAssert 值为 true 时， offset 度，默认为 false。
buf.readIntLE(offset, byteLength[, noAssert])	支持读取 48 位以下的数字。noAssert 值为 true 时， offset 度，默认为 false。
buf.readIntBE(offset, byteLength[, noAssert])	支持读取 48 位以下的数字。noAssert 值为 true 时， offset 度，默认为 false。
buf.toString([encoding][, start][, end])	根据 encoding 参数（默认是 'utf8'）返回一个解码过的 strin start (默认是 0) 和 end (默认是 buffer.length)作为取值范围。
buf.toJSON()	将 Buffer 实例转换为 JSON 对象。
buf[index]	获取或设置指定的字节。返回值代表一个字节，所以返回值 0xFF 或者十进制0至 255。
buf.equals(otherBuffer)	比较两个缓冲区是否相等，如果是返回 true，否则返回 fals
buf.compare(otherBuffer)	比较两个 Buffer 对象，返回一个数字，表示 buf 在 otherBu
buf.copy(targetBuffer[, targetStart][, sourceStart][, sourceEnd])	buffer 拷贝，源和目标可以相同。 targetStart 目标开始偏移 是 0。 sourceEnd 源结束位置偏移默认是源的长度 buffer.le
buf.slice([start][, end])	剪切 Buffer 对象，根据 start(默认是 0) 和 end (默认是 buff 负的索引是从 buffer 尾部开始计算的。

buf.readUInt8(offset[, noAssert])	根据指定的偏移量，读取一个有符号 8 位整数。若参数 noA 移量参数。 如果这样 offset 可能会超出buffer 的末尾。默认
buf.readUInt16LE(offset[, noAssert])	根据指定的偏移量，使用特殊的 endian 字节序格式读取一 noAssert 为 true 将不会验证 offset 偏移量参数。 这意味着 默认是 false。
buf.readUInt16BE(offset[, noAssert])	根据指定的偏移量，使用特殊的 endian 字节序格式读取一 noAssert 为 true 将不会验证 offset 偏移量参数。 这意味着 默认是 false。
buf.readUInt32LE(offset[, noAssert])	根据指定的偏移量，使用指定的 endian 字节序格式读取一 noAssert 为 true 将不会验证 offset 偏移量参数。 这意味着 默认是 false。
buf.readUInt32BE(offset[, noAssert])	根据指定的偏移量，使用指定的 endian 字节序格式读取一 noAssert 为 true 将不会验证 offset 偏移量参数。 这意味着 默认是 false。
buf.readInt8(offset[, noAssert])	根据指定的偏移量，读取一个 signed 8 位整数。 若参数 no 偏移量参数。 这意味着 offset 可能会超出 buffer 的末尾。默
buf.readInt16LE(offset[, noAssert])	根据指定的偏移量，使用特殊的 endian 格式读取一个 signe 为 true 将不会验证 offset 偏移量参数。 这意味着 offset 可能 false。
buf.readInt16BE(offset[, noAssert])	根据指定的偏移量，使用特殊的 endian 格式读取一个 signe 为 true 将不会验证 offset 偏移量参数。 这意味着 offset 可能 false。
buf.readInt32LE(offset[, noAssert])	根据指定的偏移量，使用指定的 endian 字节序格式读取一 noAssert 为 true 将不会验证 offset 偏移量参数。 这意味着 默认是 false。
buf.readInt32BE(offset[, noAssert])	根据指定的偏移量，使用指定的 endian 字节序格式读取一 noAssert 为 true 将不会验证 offset 偏移量参数。 这意味着 默认是 false。
buf.readFloatLE(offset[, noAssert])	根据指定的偏移量，使用指定的 endian 字节序格式读取一 为 true 将不会验证 offset 偏移量参数。 这意味着 offset 可能 false。
buf.readFloatBE(offset[, noAssert])	根据指定的偏移量，使用指定的 endian 字节序格式读取一 为 true 将不会验证 offset 偏移量参数。 这意味着 offset 可能 false。
buf.readDoubleLE(offset[, noAssert])	根据指定的偏移量，使用指定的 endian字节序格式读取一个 为 true 将不会验证 offset 偏移量参数。 这意味着 offset 可能 false。
buf.readDoubleBE(offset[, noAssert])	根据指定的偏移量，使用指定的 endian字节序格式读取一个 为 true 将不会验证 offset 偏移量参数。 这意味着 offset 可能 false。
buf.writeUInt8(value, offset[, noAssert])	根据传入的 offset 偏移量将 value 写入 buffer。注意：value 数。 若参数 noAssert 为 true 将不会验证 offset 偏移量参数 者 offset 可能会超出 buffer 的末尾从而造成 value 被丢弃。

	否则不要使用。默认是 false。
buf.writeUInt16LE(value, offset[, noAssert])	根据传入的 offset 偏移量和指定的 endian 格式将 value 写入个合法的有符号 16 位整数。 若参数 noAssert 为 true 将不会数。 这意味着 value 可能过大，或者 offset 可能会超出buff除非你对这个参数非常有把握，否则尽量不要使用。默认是
buf.writeUInt16BE(value, offset[, noAssert])	根据传入的 offset 偏移量和指定的 endian 格式将 value 写入个合法的有符号 16 位整数。 若参数 noAssert 为 true 将不会数。 这意味着 value 可能过大，或者 offset 可能会超出buff除非你对这个参数非常有把握，否则尽量不要使用。默认是
buf.writeUInt32LE(value, offset[, noAssert])	根据传入的 offset 偏移量和指定的 endian 格式将 value 写入合法的有符号 32 位整数。 若参数 noAssert 为 true 将不会这意味着value 可能过大，或者offset可能会超出buffer的末尾对这个参数非常有把握，否则尽量不要使用。默认是 false。
buf.writeUInt32BE(value, offset[, noAssert])	根据传入的 offset 偏移量和指定的 endian 格式将 value 写入合法的有符号 32 位整数。 若参数 noAssert 为 true 将不会这意味着value 可能过大，或者offset可能会超出buffer的末尾对这个参数非常有把握，否则尽量不要使用。默认是 false。
buf.writeInt8(value, offset[, noAssert])	
buf.writeInt16LE(value, offset[, noAssert])	根据传入的 offset 偏移量和指定的 endian 格式将 value 写入个合法的 signed 16 位整数。 若参数 noAssert 为 true 将不数。 这意味着 value 可能过大，或者 offset 可能会超出 buf弃。 除非你对这个参数非常有把握，否则尽量不要使用。默
buf.writeInt16BE(value, offset[, noAssert])	根据传入的 offset 偏移量和指定的 endian 格式将 value 写入个合法的 signed 16 位整数。 若参数 noAssert 为 true 将不数。 这意味着 value 可能过大，或者 offset 可能会超出 buf弃。 除非你对这个参数非常有把握，否则尽量不要使用。默
buf.writeInt32LE(value, offset[, noAssert])	根据传入的 offset 偏移量和指定的 endian 格式将 value 写入个合法的 signed 32 位整数。 若参数 noAssert 为 true 将不数。 这意味着 value 可能过大，或者 offset 可能会超出 buf弃。 除非你对这个参数非常有把握，否则尽量不要使用。默
buf.writeInt32BE(value, offset[, noAssert])	根据传入的 offset 偏移量和指定的 endian 格式将 value 写入个合法的 signed 32 位整数。 若参数 noAssert 为 true 将不数。 这意味着 value 可能过大，或者 offset 可能会超出 buf弃。 除非你对这个参数非常有把握，否则尽量不要使用。默
buf.writeFloatLE(value, offset[, noAssert])	根据传入的 offset 偏移量和指定的 endian 格式将 value 写入个 32 位浮点数类型的值时，结果将是不确定的。 若参数 no和 offset 偏移量参数。 这意味着 value可能过大，或者 offse造成 value 被丢弃。 除非你对这个参数非常有把握，否则尽
buf.writeFloatBE(value, offset[, noAssert])	根据传入的 offset 偏移量和指定的 endian 格式将 value 写入个 32 位浮点数类型的值时，结果将是不确定的。 若参数 no和 offset 偏移量参数。 这意味着 value可能过大，或者 offse造成 value 被丢弃。 除非你对这个参数非常有把握，否则尽
buf.writeDoubleLE(value,	根据传入的 offset 偏移量和指定的 endian 格式将 value 写入个有效的 64 位double 类型的值。 若参数 noAssert 为 true

offset[, noAssert])	参数。 这意味着 value 可能过大，或者 offset 可能会超出 b 弃。 除非你对这个参数非常有把握，否则尽量不要使用。默
buf.writeDoubleBE(value, offset[, noAssert])	根据传入的 offset 偏移量和指定的 endian 格式将 value 写入 个有效的 64 位double 类型的值。 若参数 noAssert 为 true 参数。 这意味着 value 可能过大，或者 offset 可能会超出 b 弃。 除非你对这个参数非常有把握，否则尽量不要使用。默
buf.fill(value[, offset][, end])	使用指定的 value 来填充这个 buffer。如果没有指定 offset (buffer.length)， 将会填充整个buffer。

Node.js Stream(流)

Stream 是一个抽象接口，Node 中有很多对象实现了这个接口。例如，对http 服务器发起请求的request 对象就是一个 Stream，还有stdout（标准输出）。

Node.js，Stream 有四种流类型：

- **Readable** - 可读操作。
- **Writable** - 可写操作。
- **Duplex** - 可读可写操作.
- **Transform** - 操作被写入数据，然后读出结果。

所有的 Stream 对象都是 EventEmitter 的实例。常用的事件有：

- **data** - 当有数据可读时触发。
- **end** - 没有更多的数据可读时触发。
- **error** - 在接收和写入过程中发生错误时触发。
- **finish** - 所有数据已被写入到底层系统时触发。

本教程会为大家介绍常用的流操作。

从流中读取数据

创建 input.txt 文件，内容如下：

```
菜鸟教程官网地址：www.runoob.com
```

创建 main.js 文件, 代码如下：

```
var fs = require("fs");
var data = '';

// 创建可读流
var readerStream = fs.createReadStream('input.txt');

// 设置编码为 utf8。
readerStream.setEncoding('UTF8');

// 处理流事件 --> data, end, and error
readerStream.on('data', function(chunk) {
   data += chunk;
});

readerStream.on('end',function(){
   console.log(data);
});

readerStream.on('error', function(err){
   console.log(err.stack);
});

console.log("程序执行完毕");
```

以上代码执行结果如下：

```
程序执行完毕
菜鸟教程官网地址：www.runoob.com
```

写入流

创建 main.js 文件, 代码如下：

```
var fs = require("fs");
var data = '菜鸟教程官网地址：www.runoob.com';

// 创建一个可以写入的流，写入到文件 output.txt 中
var writerStream = fs.createWriteStream('output.txt');

// 使用 utf8 编码写入数据
writerStream.write(data,'UTF8');

// 标记文件末尾
writerStream.end();

// 处理流事件 --> data, end, and error
writerStream.on('finish', function() {
    console.log("写入完成。");
});

writerStream.on('error', function(err){
   console.log(err.stack);
});

console.log("程序执行完毕");
```

以上程序会将 data 变量的数据写入到 output.txt 文件中。代码执行结果如下：

```
$ node main.js
程序执行完毕
写入完成。
```

查看 output.txt 文件的内容：

```
$ cat output.txt
菜鸟教程官网地址：www.runoob.com
```

管道流

管道提供了一个输出流到输入流的机制。通常我们用于从一个流中获取数据并将数据传递到另外一个流中。

如上面的图片所示，我们把文件比作装水的桶，而水就是文件里的内容，我们用一根管子(pipe)连接两个桶使得水从一个桶流入另一个桶，这样就慢慢的实现了大文件的复制过程。

以下实例我们通过读取一个文件内容并将内容写入到另外一个文件中。

设置 input.txt 文件内容如下：

```
菜鸟教程官网地址：www.runoob.com
管道流操作实例
```

创建 main.js 文件, 代码如下：

```
var fs = require("fs");

// 创建一个可读流
var readerStream = fs.createReadStream('input.txt');

// 创建一个可写流
var writerStream = fs.createWriteStream('output.txt');

// 管道读写操作
// 读取 input.txt 文件内容，并将内容写入到 output.txt 文件中
readerStream.pipe(writerStream);

console.log("程序执行完毕");
```

代码执行结果如下：

```
$ node main.js
程序执行完毕
```

查看 output.txt 文件的内容：

```
$ cat output.txt
菜鸟教程官网地址：www.runoob.com
管道流操作实例
```

链式流

链式是通过连接输出流到另外一个流并创建多个对个流操作链的机制。链式流一般用于管道操作。

接下来我们就是用管道和链式来压缩和解压文件。

创建 compress.js 文件, 代码如下：

```
var fs = require("fs");
var zlib = require('zlib');

// 压缩 input.txt 文件为 input.txt.gz
fs.createReadStream('input.txt')
  .pipe(zlib.createGzip())
  .pipe(fs.createWriteStream('input.txt.gz'));

console.log("文件压缩完成。");
```

代码执行结果如下：

```
$ node compress.js
文件压缩完成。
```

执行完以上操作后，我们可以看到当前目录下生成了 input.txt 的压缩文件 input.txt.gz。

接下来，让我们来解压该文件，创建 decompress.js 文件，代码如下：

```
var fs = require("fs");
var zlib = require('zlib');

// 解压 input.txt.gz 文件为 input.txt
fs.createReadStream('input.txt.gz')
  .pipe(zlib.createGunzip())
  .pipe(fs.createWriteStream('input.txt'));

console.log("文件解压完成。");
```

代码执行结果如下：

```
$ node decompress.js
文件解压完成。
```

Node.js模块系统

为了让Node.js的文件可以相互调用，Node.js提供了一个简单的模块系统。

模块是Node.js 应用程序的基本组成部分，文件和模块是一一对应的。换言之，一个 Node.js 文件就是一个模块，这个文件可能是JavaScript 代码、JSON 或者编译过的C/C++ 扩展。

创建模块

在 Node.js 中，创建一个模块非常简单，如下我们创建一个 'main.js' 文件，代码如下:

```
var hello = require('./hello');
hello.world();
```

以上实例中，代码 require('./hello') 引入了当前目录下的hello.js文件（./ 为当前目录，node.js 默认后缀为js）。

Node.js 提供了exports 和 require 两个对象，其中 exports 是模块公开的接口，require 用于从外部获取一个模块的接口，即所获取模块的 exports 对象。

接下来我们就来创建hello.js文件，代码如下：

```
exports.world = function() {
  console.log('Hello World');
}
```

在以上示例中，hello.js 通过 exports 对象把 world 作为模块的访 问接口，在 main.js 中通过 require('./hello') 加载这个模块，然后就可以直接访 问main.js 中 exports 对象的成员函数了。

有时候我们只是想把一个对象封装到模块中，格式如下：

```
module.exports = function() {
  // ...
}
```

例如:

```
//hello.js
function Hello() {
    varname;
    this.setName = function(thyName) {
        name = thyName;
    };
    this.sayHello = function() {
        console.log('Hello ' + name);
    };
};
module.exports = Hello;
```

这样就可以直接获得这个对象了：

```
//main.js
var Hello = require('./hello');
hello = new Hello();
hello.setName('BYVoid');
hello.sayHello();
```

模块接口的唯一变化是使用 module.exports = Hello 代替了exports.world = function(){}。 在外部引用该模块时，其接口对象就是要输出的 Hello 对象本身，而不是原先的 exports。

服务端的模块放在哪里

也许你已经注意到，我们已经在代码中使用了模块了。像这样：

```
var http = require("http");

...

http.createServer(...);
```

Node.js中自带了一个叫做"http"的模块，我们在我们的代码中请求它并把返回值赋给一个本地变量。

这把我们的本地变量变成了一个拥有所有 http 模块所提供的公共方法的对象。

Node.js 的 require方法中的文件查找策略如下：

由于Node.js中存在4类模块（原生模块和3种文件模块），尽管require方法极其简单，但是内部的加载却是十分复杂的，其加载优先级也各自不同。如下图所示：

从文件模块缓存中加载

尽管原生模块与文件模块的优先级不同，但是都不会优先于从文件模块的缓存中加载已经存在的模块。

从原生模块加载

原生模块的优先级仅次于文件模块缓存的优先级。require方法在解析文件名之后，优先检查模块是否在原生模块列表中。以http模块为例，尽管在目录下存在一个http/http.js/http.node/http.json文件，require("http")都不会从这些文件中加载，而是从原生模块中加载。

原生模块也有一个缓存区，同样也是优先从缓存区加载。如果缓存区没有被加载过，则调用原生模块的加载方式进行加载和执行。

从文件加载

当文件模块缓存中不存在，而且不是原生模块的时候，Node.js会解析require方法传入的参数，并从文件系统中加载实际的文件，加载过程中的包装和编译细节在前一节中已经介绍过，这里我们将详细描述查找文件模块的过程，其中，也有一些细节值得知晓。

require方法接受以下几种参数的传递：

- http、fs、path等，原生模块。
- ./mod或../mod，相对路径的文件模块。
- /pathtomodule/mod，绝对路径的文件模块。
- mod，非原生模块的文件模块。

Node.js 函数

在JavaScript中，一个函数可以作为另一个函数接收一个参数。我们可以先定义一个函数，然后传递，也可以在传递参数的地方直接定义函数。

Node.js中函数的使用与Javascript类似，举例来说，你可以这样做：

```
function say(word) {
  console.log(word);
}

function execute(someFunction, value) {
  someFunction(value);
}

execute(say, "Hello");
```

以上代码中，我们把 say 函数作为execute函数的第一个变量进行了传递。这里返回的不是 say 的返回值，而是 say 本身！

这样一来， say 就变成了execute 中的本地变量 someFunction ， execute可以通过调用 someFunction() （带括号的形式）来使用 say 函数。

当然，因为 say 有一个变量， execute 在调用 someFunction 时可以传递这样一个变量。

匿名函数

我们可以把一个函数作为变量传递。但是我们不一定要绕这个"先定义，再传递"的圈子，我们可以直接在另一个函数的括号中定义和传递这个函数：

```
function execute(someFunction, value) {
  someFunction(value);
}

execute(function(word){ console.log(word) }, "Hello");
```

我们在 execute 接受第一个参数的地方直接定义了我们准备传递给 execute 的函数。

用这种方式，我们甚至不用给这个函数起名字，这也是为什么它被叫做匿名函数 。

函数传递是如何让HTTP服务器工作的

带着这些知识，我们再来看看我们简约而不简单的HTTP服务器：

```
var http = require("http");

http.createServer(function(request, response) {
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.write("Hello World");
  response.end();
}).listen(8888);
```

现在它看上去应该清晰了很多：我们向 createServer 函数传递了一个匿名函数。

用这样的代码也可以达到同样的目的：

```
var http = require("http");

function onRequest(request, response) {
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.write("Hello World");
  response.end();
}

http.createServer(onRequest).listen(8888);
```

Node.js 路由

我们要为路由提供请求的URL和其他需要的GET及POST参数，随后路由需要根据这些数据来执行相应的代码。

因此，我们需要查看HTTP请求，从中提取出请求的URL以及GET/POST参数。这一功能应当属于路由还是服务器（甚至作为一个模块自身的功能）确实值得探讨，但这里暂定其为我们的HTTP服务器的功能。

我们需要的所有数据都会包含在request对象中，该对象作为onRequest()回调函数的第一个参数传递。但是为了解析这些数据，我们需要额外的Node.JS模块，它们分别是url和querystring模块。

```
                   url.parse(string).query
                                           |
           url.parse(string).pathname      |
                       |                   |
                       |                   |
                     ------ -------------------
http://localhost:8888/start?foo=bar&hello=world
                                ---       -----
                                 |          |
                                 |          |
              querystring(string)["foo"]    |
                                            |
                         querystring(string)["hello"]
```

当然我们也可以用querystring模块来解析POST请求体中的参数，稍后会有演示。

现在我们来给onRequest()函数加上一些逻辑，用来找出浏览器请求的URL路径：

```
var http = require("http");
var url = require("url");

function start() {
  function onRequest(request, response) {
    var pathname = url.parse(request.url).pathname;
    console.log("Request for " + pathname + " received.");
    response.writeHead(200, {"Content-Type": "text/plain"});
    response.write("Hello World");
    response.end();
  }

  http.createServer(onRequest).listen(8888);
  console.log("Server has started.");
}

exports.start = start;
```

好了，我们的应用现在可以通过请求的URL路径来区别不同请求了--这使我们得以使用路由（还未完成）来将请求以URL路径为基准映射到处理程序上。

在我们所要构建的应用中，这意味着来自/start和/upload的请求可以使用不同的代码来处理。稍后我们将看到这些内容是如何整合到一起的。

现在我们可以来编写路由了，建立一个名为router.js的文件，添加以下内容：

```
function route(pathname) {
  console.log("About to route a request for " + pathname);
}

exports.route = route;
```

如你所见，这段代码什么也没干，不过对于现在来说这是应该的。在添加更多的逻辑以前，我们先来看看如何把路由和服务器整合起来。

我们的服务器应当知道路由的存在并加以有效利用。我们当然可以通过硬编码的方式将这一依赖项绑定到服务器上，但是其它语言的编程经验告诉我们这会是一件非常痛苦的事，因此我们将使用依赖注入的方式较松散地添加路由模块。

首先，我们来扩展一下服务器的start()函数，以便将路由函数作为参数传递过去：

```
var http = require("http");
var url = require("url");

function start(route) {
  function onRequest(request, response) {
    var pathname = url.parse(request.url).pathname;
    console.log("Request for " + pathname + " received.");

    route(pathname);

    response.writeHead(200, {"Content-Type": "text/plain"});
    response.write("Hello World");
    response.end();
  }

  http.createServer(onRequest).listen(8888);
  console.log("Server has started.");
}

exports.start = start;
```

同时，我们会相应扩展index.js，使得路由函数可以被注入到服务器中：

```
var server = require("./server");
var router = require("./router");

server.start(router.route);
```

在这里，我们传递的函数依旧什么也没做。

如果现在启动应用（node index.js，始终记得这个命令行），随后请求一个URL，你将会看到应用输出相应的信息，这表明我们的HTTP服务器已经在使用路由模块了，并会将请求的路径传递给路由：

```
bash$ node index.js
Request for /foo received.
About to route a request for /foo
```

以上输出已经去掉了比较烦人的/favicon.ico请求相关的部分。

Node.js 全局对象

JavaScript 中有一个特殊的对象，称为全局对象（Global Object），它及其所有属性都可 以在程序的任何地方访问，即全局变量。

在浏览器JavaScript 中，通常window 是全局对象， 而Node.js 中的全局对象是 global，所有全局变量（除了 global 本身以外）都是 global 对象的属性。

我们在Node.js 中能够直接访问到对象通常都是 global 的属性，如 console、process 等，下面逐一介绍。

全局对象与全局变量

global 最根本的作用是作为全局变量的宿主。按照ECMAScript 的定义，满足以下条 件的变量是全局变量：

- 在最外层定义的变量；
- 全局对象的属性；
- 隐式定义的变量（未定义直接赋值的变量）。

当你定义一个全局变量时，这个变量同时也会成为全局对象的属性，反之亦然。需要注 意的是，在Node.js 中你不可能在最外层定义变量，因为所有用户代码都是属于当前模块的， 而模块本身不是最外层上下文。

注意： 永远使用var 定义变量以避免引入全局变量，因为全局变量会污染 命名空间，提高代码的耦合风险。

process

process 是一个全局变量，即 global 对象的属性。

它用于描述当前Node.js 进程状态 的对象，提供了一个与操作系统的简单接口。通常在你写本地命令行程序的时候，少不了要 和它打交道。下面将会介绍process 对象的一些最常用的成员方法。

process.argv是命令行参数数组，第一个元素是 node，第二个元素是脚本文件名， 从第三个元素开始每个元素是一个运行参数。

```
console.log(process.argv);
```

将以上代码存储为argv.js，通过以下命令运行：

```
$ node argv.js 1991 name=byvoid --v "Carbo Kuo"
[ 'node',
'/home/byvoid/argv.js',
'1991',
'name=byvoid',
'--v',
'Carbo Kuo' ]
```

- **process.stdout**是标准输出流，通常我们使用的 console.log() 向标准输出打印 字符，而 process.stdout.write() 函数提供了更底层的接口。
- **process.stdin**是标准输入流，初始时它是被暂停的，要想从标准输入读取数据， 你必须恢复流，并手动编写流的事件响应函数。

```
process.stdin.resume();
process.stdin.on('data', function(data) {
process.stdout.write('read from console: ' + data.toString());
});
```

- **process.nextTick(callback)**的功能是为事件循环设置一项任务，Node.js 会在 下次事件循环调响应时调用 callback。

初学者很可能不理解这个函数的作用，有什么任务不能在当下执行完，需要交给下次事 件循环响应来做呢？

我们讨论过，Node.js 适合I/O 密集型的应用，而不是计算密集型的应用， 因为一个Node.js 进程只有一个线程，因此在任何时刻都只有一个事件在执行。

如果这个事 件占用大量的CPU 时间，执行事件循环中的下一个事件就需要等待很久，因此 Node.js 的一 个编程原则就是尽量缩短每个事件的执行时间。process.nextTick() 提供了一个这样的 工具，可以把复杂的工作拆散， 变成一个个较小的事件。

```
functiondoSomething(args, callback) {
    somethingComplicated(args);
    callback();
}
doSomething(functiononEnd() {
    compute();
});
```

我们假设compute() 和somethingComplicated() 是两个较为耗时的函数，以上 的程序在调用 doSomething() 时会先执行somethingComplicated()，然后立即调用 回调函数，在 onEnd() 中又会执行 compute()。下面用process.nextTick() 改写上 面的程序：

```
functiondoSomething(args, callback) {
    somethingComplicated(args);
    process.nextTick(callback);
}
doSomething(functiononEnd() {
    compute();
});
```

改写后的程序会把上面耗时的操作拆分为两个事件，减少每个事件的执行时间，提高事 件响应速度。

注意： 不要使用setTimeout(fn,0)代替process.nextTick(callback)， 前者比后者效率要低得多。

我们探讨了process对象常用的几个成员，除此之外process还展示了process.platform、process.pid、process.execPath、process.memoryUsage() 等方法，以及POSIX 进程信号响应机制。有兴趣的读者可以访问http://nodejs.org/api/process.html 了解详细 内容。

console

console 用于提供控制台标准输出，它是由Internet Explorer 的JScript 引擎提供的调试 工具，后来逐渐成为浏览器的事实标准。

Node.js 沿用了这个标准，提供与习惯行为一致的 console 对象，用于向标准输出流（stdout）或标准错误流（stderr）输出字符。 ? console.log()：向标准输出流打印字符并以换行符结束。

console.log 接受若干 个参数，如果只有一个参数，则输出这个参数的字符串形式。如果有多个参数，则 以类似于C 语言 printf() 命令的格式输出。

第一个参数是一个字符串，如果没有 参数，只打印一个换行。

```
console.log('Hello world');
console.log('byvoid%diovyb');
console.log('byvoid%diovyb', 1991);
```

运行结果为：

```
Hello world
byvoid%diovyb
byvoid1991iovyb
```

- console.error()：与console.log() 用法相同，只是向标准错误流输出。
- console.trace()：向标准错误流输出当前的调用栈。

```
console.trace();
```

运行结果为：

```
Trace:
at Object.<anonymous> (/home/byvoid/consoletrace.js:1:71)
at Module._compile (module.js:441:26)
at Object..js (module.js:459:10)
at Module.load (module.js:348:31)
at Function._load (module.js:308:12)
at Array.0 (module.js:479:10)
at EventEmitter._tickCallback (node.js:192:40)
```

Node.js 常用工具 util

util 是一个Node.js 核心模块，提供常用函数的集合，用于弥补核心JavaScript 的功能 过于精简的不足。

util.inherits

util.inherits(constructor, superConstructor)是一个实现对象间原型继承 的函数。

JavaScript 的面向对象特性是基于原型的，与常见的基于类的不同。JavaScript 没有 提供对象继承的语言级别特性，而是通过原型复制来实现的。

在这里我们只介绍util.inherits 的用法，示例如下：

```
varutil = require('util');
functionBase() {
    this.name = 'base';
    this.base = 1991;
    this.sayHello = function() {
    console.log('Hello ' + this.name);
    };
}
Base.prototype.showName = function() {
    console.log(this.name);
};
functionSub() {
    this.name = 'sub';
}
util.inherits(Sub, Base);
varobjBase = newBase();
objBase.showName();
objBase.sayHello();
console.log(objBase);
varobjSub = newSub();
objSub.showName();
//objSub.sayHello();
console.log(objSub);
```

我们定义了一个基础对象Base 和一个继承自Base 的Sub，Base 有三个在构造函数 内定义的属性和一个原型中定义的函数，通过util.inherits 实现继承。运行结果如下：

```
base
Hello base
{ name: 'base', base: 1991, sayHello: [Function] }
sub
{ name: 'sub' }
```

注意：Sub 仅仅继承了Base 在原型中定义的函数，而构造函数内部创造的 base 属 性和 sayHello 函数都没有被 Sub 继承。

同时，在原型中定义的属性不会被console.log 作 为对象的属性输出。如果我们去掉 objSub.sayHello(); 这行的注释，将会看到：

```
node.js:201
throw e; // process.nextTick error, or 'error' event on first tick
^
TypeError: Object #<Sub> has no method 'sayHello'
at Object.<anonymous> (/home/byvoid/utilinherits.js:29:8)
at Module._compile (module.js:441:26)
at Object..js (module.js:459:10)
at Module.load (module.js:348:31)
at Function._load (module.js:308:12)
at Array.0 (module.js:479:10)
at EventEmitter._tickCallback (node.js:192:40)
```

util.inspect

util.inspect(object,[showHidden],[depth],[colors])是一个将任意对象转换 为字符串的方法，通常用于调试和错误输出。它至少接受一个参数 object，即要转换的对象。

showHidden 是一个可选参数，如果值为 true，将会输出更多隐藏信息。

depth 表示最大递归的层数，如果对象很复杂，你可以指定层数以控制输出信息的多 少。如果不指定depth，默认会递归2层，指定为 null 表示将不限递归层数完整遍历对象。 如果color 值为 true，输出格式将会以ANSI 颜色编码，通常用于在终端显示更漂亮 的效果。

特别要指出的是，util.inspect 并不会简单地直接把对象转换为字符串，即使该对 象定义了 toString 方法也不会调用。

```
varutil = require('util');
functionPerson() {
    this.name = 'byvoid';
    this.toString = function() {
    return this.name;
    };
}
varobj = newPerson();
console.log(util.inspect(obj));
console.log(util.inspect(obj, true));
```

运行结果是：

```
{ name: 'byvoid', toString: [Function] }
{ toString:
{ [Function]
[prototype]: { [constructor]: [Circular] },
[caller]: null,
[length]: 0,
[name]: '',
[arguments]: null },
name: 'byvoid' }
```

util.isArray(object)

如果给定的参数 "object" 是一个数组返回true，否则返回false。

```
var util = require('util');

util.isArray([])
  // true
util.isArray(new Array)
  // true
util.isArray({})
  // false
```

util.isRegExp(object)

如果给定的参数 "object" 是一个正则表达式返回true，否则返回false。

```
var util = require('util');

util.isRegExp(/some regexp/)
  // true
util.isRegExp(new RegExp('another regexp'))
  // true
util.isRegExp({})
  // false
```

util.isDate(object)

如果给定的参数 "object" 是一个日期返回true，否则返回false。

```
var util = require('util');

util.isDate(new Date())
  // true
util.isDate(Date())
  // false (without 'new' returns a String)
util.isDate({})
  // false
```

util.isError(object)

如果给定的参数 "object" 是一个错误对象返回true，否则返回false。

```
var util = require('util');

util.isError(new Error())
  // true
util.isError(new TypeError())
  // true
util.isError({ name: 'Error', message: 'an error occurred' })
  // false
```

更多详情可以访问 http://nodejs.org/api/util.html 了解详细内容。

Node.js 文件系统

Node.js 提供一组类似 UNIX（POSIX）标准的文件操作API。 Node 导入文件系统模块(fs)语法如下所示：

```
var fs = require("fs")
```

异步和同步

Node.js 文件系统（fs 模块）模块中的方法均有异步和同步版本，例如读取文件内容的函数有异步的 fs.readFile() 和同步的 fs.readFileSync()。

异步的方法函数最后一个参数为回调函数，回调函数的第一个参数包含了错误信息(error)。

建议大家是用异步方法，比起同步，异步方法性能更高，速度更快，而且没有阻塞。

实例

创建 input.txt 文件，内容如下：

```
菜鸟教程官网地址：www.runoob.com
文件读取实例
```

创建 file.js 文件, 代码如下：

```
var fs = require("fs");

// 异步读取
fs.readFile('input.txt', function (err, data) {
   if (err) {
       return console.error(err);
   }
   console.log("异步读取: " + data.toString());
});

// 同步读取
var data = fs.readFileSync('input.txt');
console.log("同步读取: " + data.toString());

console.log("程序执行完毕。");
```

以上代码执行结果如下：

```
$ node file.js
同步读取: 菜鸟教程官网地址：www.runoob.com
文件读取实例

程序执行完毕。
异步读取: 菜鸟教程官网地址：www.runoob.com
文件读取实例
```

接下来，让我们来具体了解下 Node.js 文件系统的方法。

打开文件

语法

以下为在异步模式下打开文件的语法格式：

```
fs.open(path, flags[, mode], callback)
```

参数

参数使用说明如下：

- **path** - 文件的路径。
- **flags** - 文件打开的行为。具体值详见下文。
- **mode** - 设置文件模式(权限)，文件创建默认权限为 0666(可读，可写)。
- **callback** - 回调函数，带有两个参数如：callback(err, fd)。

flags 参数可以是以下值：

Flag	描述
r	以读取模式打开文件。如果文件不存在抛出异常。
r+	以读写模式打开文件。如果文件不存在抛出异常。
rs	以同步的方式读取文件。
rs+	以同步的方式读取和写入文件。
w	以写入模式打开文件，如果文件不存在则创建。
wx	类似 'w'，但是如果文件路径不存在，则文件写入失败。
w+	以读写模式打开文件，如果文件不存在则创建。
wx+	类似 'w+'， 但是如果文件路径不存在，则文件读写失败。
a	以追加模式打开文件，如果文件不存在则创建。
ax	类似 'a'， 但是如果文件路径不存在，则文件追加失败。
a+	以读取追加模式打开文件，如果文件不存在则创建。
ax+	类似 'a+'， 但是如果文件路径不存在，则文件读取追加失败。

实例

接下来我们创建 file.js 文件，并打开 input.txt 文件进行读写，代码如下所示：

```
var fs = require("fs");

// 异步打开文件
console.log("准备打开文件！");
fs.open('input.txt', 'r+', function(err, fd) {
   if (err) {
       return console.error(err);
   }
  console.log("文件打开成功！");
});
```

以上代码执行结果如下：

```
$ node file.js
准备打开文件！
文件打开成功！
```

获取文件信息

语法

以下为通过异步模式获取文件信息的语法格式：

```
fs.stat(path, callback)
```

参数

参数使用说明如下：

- **path** - 文件路径。
- **callback** - 回调函数，带有两个参数如：(err, stats), **stats** 是 fs.Stats 对象。

fs.stat(path)执行后，会将stats类的实例返回给其回调函数。可以通过stats类中的提供方法判断文件的相关属性。例如判断是否为文件：

```
var fs = require('fs');

fs.stat('/Users/liuht/code/itbilu/demo/fs.js', function (err, stats) {
    console.log(stats.isFile());         //true
})
```

stats类中的方法有：

方法	描述
stats.isFile()	如果是文件返回 true，否则返回 false。
stats.isDirectory()	如果是目录返回 true，否则返回 false。
stats.isBlockDevice()	如果是块设备返回 true，否则返回 false。
stats.isCharacterDevice()	如果是字符设备返回 true，否则返回 false。
stats.isSymbolicLink()	如果是软链接返回 true，否则返回 false。
stats.isFIFO()	如果是FIFO，返回true，否则返回 false。FIFO是UNIX中的一种特殊类型的命令管道。
stats.isSocket()	如果是 Socket 返回 true，否则返回 false。

实例

接下来我们创建 file.js 文件，代码如下所示：

```
var fs = require("fs");

console.log("准备打开文件！");
fs.stat('input.txt', function (err, stats) {
   if (err) {
       return console.error(err);
   }
   console.log(stats);
   console.log("读取文件信息成功！");

   // 检测文件类型
   console.log("是否为文件(isFile) ? " + stats.isFile());
   console.log("是否为目录(isDirectory) ? " + stats.isDirectory());
});
```

以上代码执行结果如下：

```
$ node file.js
准备打开文件！
{ dev: 16777220,
  mode: 33188,
  nlink: 1,
  uid: 501,
  gid: 20,
  rdev: 0,
  blksize: 4096,
  ino: 40333161,
  size: 61,
  blocks: 8,
  atime: Mon Sep 07 2015 17:43:55 GMT+0800 (CST),
  mtime: Mon Sep 07 2015 17:22:35 GMT+0800 (CST),
  ctime: Mon Sep 07 2015 17:22:35 GMT+0800 (CST) }
读取文件信息成功！
是否为文件(isFile) ? true
是否为目录(isDirectory) ? false
```

写入文件

语法

以下为异步模式下写入文件的语法格式：

```
fs.writeFile(filename, data[, options], callback)
```

如果文件存在，该方法写入的内容会覆盖旧的文件内容。

参数

参数使用说明如下：

- **path** - 文件路径。
- **data** - 要写入文件的数据，可以是 String(字符串) 或 Buffer(流) 对象。

- **options** - 该参数是一个对象，包含 {encoding, mode, flag}。默认编码为 utf8, 模式为 0666 ， flag 为 'w'
- **callback** - 回调函数，回调函数只包含错误信息参数(err)，在写入失败时返回。

实例

接下来我们创建 file.js 文件，代码如下所示：

```
var fs = require("fs");

console.log("准备写入文件");
fs.writeFile('input.txt', '我是通过写入的文件内容！',  function(err) {
   if (err) {
       return console.error(err);
   }
   console.log("数据写入成功！");
   console.log("--------我是分割线-------------")
   console.log("读取写入的数据！");
   fs.readFile('input.txt', function (err, data) {
      if (err) {
         return console.error(err);
      }
      console.log("异步读取文件数据: " + data.toString());
   });
});
```

以上代码执行结果如下：

```
$ node file.js
准备写入文件
数据写入成功！
--------我是分割线-------------
读取写入的数据！
异步读取文件数据: 我是通过写入的文件内容
```

读取文件

语法

以下为异步模式下读取文件的语法格式：

```
fs.read(fd, buffer, offset, length, position, callback)
```

该方法使用了文件描述符来读取文件。

参数

参数使用说明如下：

- **fd** - 通过 fs.open() 方法返回的文件描述符。
- **buffer** - 数据写入的缓冲区。
- **offset** - 缓冲区写入的写入偏移量。
- **length** - 要从文件中读取的字节数。
- **position** - 文件读取的起始位置，如果 position 的值为 null，则会从当前文件指针的位置读取。
- **callback** - 回调函数，有三个参数err, bytesRead, buffer，err 为错误信息， bytesRead 表示读取的字节数，buffer 为缓冲区对象。

实例

input.txt 文件内容为：

```
菜鸟教程官网地址：www.runoob.com
```

接下来我们创建 file.js 文件，代码如下所示：

```
var fs = require("fs");
var buf = new Buffer(1024);

console.log("准备打开已存在的文件！");
fs.open('input.txt', 'r+', function(err, fd) {
   if (err) {
       return console.error(err);
   }
   console.log("文件打开成功！");
   console.log("准备读取文件：");
   fs.read(fd, buf, 0, buf.length, 0, function(err, bytes){
      if (err){
         console.log(err);
      }
      console.log(bytes + "  字节被读取");

      // 仅输出读取的字节
      if(bytes > 0){
         console.log(buf.slice(0, bytes).toString());
      }
   });
});
```

以上代码执行结果如下：

```
$ node file.js
准备打开已存在的文件！
文件打开成功！
准备读取文件：
42  字节被读取
菜鸟教程官网地址：www.runoob.com
```

关闭文件

语法

以下为异步模式下关闭文件的语法格式：

```
fs.close(fd, callback)
```

该方法使用了文件描述符来读取文件。

参数

参数使用说明如下：

- **fd** - 通过 fs.open() 方法返回的文件描述符。
- **callback** - 回调函数，没有参数。

实例

input.txt 文件内容为：

```
菜鸟教程官网地址：www.runoob.com
```

接下来我们创建 file.js 文件，代码如下所示：

```
var fs = require("fs");
var buf = new Buffer(1024);

console.log("准备打开文件！");
fs.open('input.txt', 'r+', function(err, fd) {
   if (err) {
       return console.error(err);
   }
   console.log("文件打开成功！");
   console.log("准备读取文件！");
   fs.read(fd, buf, 0, buf.length, 0, function(err, bytes){
      if (err){
         console.log(err);
      }

      // 仅输出读取的字节
      if(bytes > 0){
         console.log(buf.slice(0, bytes).toString());
      }

      // 关闭文件
      fs.close(fd, function(err){
         if (err){
            console.log(err);
         }
         console.log("文件关闭成功");
      });
   });
});
```

以上代码执行结果如下：

```
$ node file.js
准备打开文件！
文件打开成功！
准备读取文件！
菜鸟教程官网地址：www.runoob.com
文件关闭成功
```

截取文件

语法

以下为异步模式下截取文件的语法格式：

```
fs.ftruncate(fd, len, callback)
```

该方法使用了文件描述符来读取文件。

参数

参数使用说明如下：

- **fd** - 通过 fs.open() 方法返回的文件描述符。

- **len** - 文件内容截取的长度。
- **callback** - 回调函数，没有参数。

实例

input.txt 文件内容为：

```
site:www.runoob.com
```

接下来我们创建 file.js 文件，代码如下所示：

```
var fs = require("fs");
var buf = new Buffer(1024);

console.log("准备打开文件！");
fs.open('input.txt', 'r+', function(err, fd) {
   if (err) {
       return console.error(err);
   }
   console.log("文件打开成功！");
   console.log("截取10字节后的文件内容。");

   // 截取文件
   fs.ftruncate(fd, 10, function(err){
      if (err){
         console.log(err);
      }
      console.log("文件截取成功。");
      console.log("读取相同的文件");
      fs.read(fd, buf, 0, buf.length, 0, function(err, bytes){
         if (err){
            console.log(err);
         }

         // 仅输出读取的字节
         if(bytes > 0){
            console.log(buf.slice(0, bytes).toString());
         }

         // 关闭文件
         fs.close(fd, function(err){
            if (err){
               console.log(err);
            }
            console.log("文件关闭成功！");
         });
      });
   });
});
```

以上代码执行结果如下：

```
$ node file.js
准备打开文件！
文件打开成功！
截取10字节后的文件内容。
文件截取成功。
读取相同的文件
site:www.r
文件关闭成功
```

删除文件

语法

以下为删除文件的语法格式：

```
fs.unlink(path, callback)
```

参数

参数使用说明如下：

- **path** - 文件路径。
- **callback** - 回调函数，没有参数。

实例

input.txt 文件内容为：

```
site:www.runoob.com
```

接下来我们创建 file.js 文件，代码如下所示：

```
var fs = require("fs");

console.log("准备删除文件！");
fs.unlink('input.txt', function(err) {
   if (err) {
       return console.error(err);
   }
   console.log("文件删除成功！");
});
```

以上代码执行结果如下：

```
$ node file.js
准备删除文件！
文件删除成功！
```

再去查看 input.txt 文件，发现已经不存在了。

创建目录

语法

以下为创建目录的语法格式：

```
fs.mkdir(path[, mode], callback)
```

参数

参数使用说明如下：

- **path** - 文件路径。
- **mode** - 设置目录权限，默认为 0777。
- **callback** - 回调函数，没有参数。

实例

接下来我们创建 file.js 文件，代码如下所示：

```
var fs = require("fs");

console.log(创建目录 /tmp/test");
fs.mkdir('/tmp/test',function(err){
   if (err) {
       return console.error(err);
   }
   console.log("目录创建成功。");
});
```

以上代码执行结果如下：

```
$ node file.js
创建目录 /tmp/test
目录创建成功。
```

读取目录

语法

以下为读取目录的语法格式：

```
fs.readdir(path, callback)
```

参数

参数使用说明如下：

- **path** - 文件路径。
- **callback** - 回调函数，回调函数带有两个参数err, files，err 为错误信息，files 为 目录下的文件数组列表。

实例

接下来我们创建 file.js 文件，代码如下所示：

```
var fs = require("fs");

console.log("查看 /tmp 目录");
fs.readdir("/tmp/",function(err, files){
   if (err) {
       return console.error(err);
   }
   files.forEach( function (file){
       console.log( file );
   });
});
```

以上代码执行结果如下：

```
$ node file.js
查看 /tmp 目录
input.out
output.out
test
test.txt
```

删除目录

语法

以下为删除目录的语法格式：

```
fs.rmdir(path, callback)
```

参数

参数使用说明如下：

- **path** - 文件路径。
- **callback** - 回调函数，没有参数。

实例

接下来我们创建 file.js 文件，代码如下所示：

```
var fs = require("fs");

console.log("准备删除目录 /tmp/test");
fs.rmdir("/tmp/test",function(err){
   if (err) {
       return console.error(err);
   }
   console.log("读取 /tmp 目录");
   fs.readdir("/tmp/",function(err, files){
      if (err) {
          return console.error(err);
      }
      files.forEach( function (file){
          console.log( file );
      });
   });
});
```

以上代码执行结果如下：

```
$ node file.js
准备删除目录 /tmp/test
input.out
output.out
test
test.txt
读取 /tmp 目录
……
```

文件模块方法参考手册

以下为 Node.js 文件模块相同的方法列表：

方法	描述
fs.rename(oldPath, newPath, callback)	异步 rename().回调函数没有参数，但可能抛出异常。
fs.ftruncate(fd, len, callback)	异步 ftruncate().回调函数没有参数，但可能抛出异常。
fs.ftruncateSync(fd, len)	同步 ftruncate()

fs.truncate(path, len, callback)	异步 truncate().回调函数没有参数，但可能抛出异常。
fs.truncateSync(path, len)	同步 truncate()
fs.chown(path, uid, gid, callback)	异步 chown().回调函数没有参数，但可能抛出异常。
fs.chownSync(path, uid, gid)	同步 chown()
fs.fchown(fd, uid, gid, callback)	异步 fchown().回调函数没有参数，但可能抛出异常。
fs.fchownSync(fd, uid, gid)	同步 fchown()
fs.lchown(path, uid, gid, callback)	异步 lchown().回调函数没有参数，但可能抛出异常。
fs.lchownSync(path, uid, gid)	同步 lchown()
fs.chmod(path, mode, callback)	异步 chmod().回调函数没有参数，但可能抛出异常。
fs.chmodSync(path, mode)	同步 chmod().
fs.fchmod(fd, mode, callback)	异步 fchmod().回调函数没有参数，但可能抛出异常。
fs.fchmodSync(fd, mode)	同步 fchmod().
fs.lchmod(path, mode, callback)	异步 lchmod().回调函数没有参数，但可能抛出异常。Only available on Mac OS X.
fs.lchmodSync(path, mode)	同步 lchmod().
fs.stat(path, callback)	异步 stat(). 回调函数有两个参数 err, stats，stats 是 fs.Stats 对象。
fs.lstat(path, callback)	异步 lstat(). 回调函数有两个参数 err, stats，stats 是 fs.Stats 对象。
fs.fstat(fd, callback)	异步 fstat(). 回调函数有两个参数 err, stats，stats 是 fs.Stats 对象。
fs.statSync(path)	同步 stat(). 返回 fs.Stats 的实例。
fs.lstatSync(path)	同步 lstat(). 返回 fs.Stats 的实例。
fs.fstatSync(fd)	同步 fstat(). 返回 fs.Stats 的实例。
fs.link(srcpath, dstpath, callback)	异步 link().回调函数没有参数，但可能抛出异常。
fs.linkSync(srcpath, dstpath)	同步 link().
fs.symlink(srcpath, dstpath[, type], callback)	异步 symlink().回调函数没有参数，但可能抛出异常。type 参数可以设置为 'dir', 'file', 或 'junction' (默认为

dstpath[, type], callback)	'file')。
fs.symlinkSync(srcpath, dstpath[, type])	同步 symlink().
fs.readlink(path, callback)	异步 readlink(). 回调函数有两个参数 err, linkString。
fs.realpath(path[, cache], callback)	异步 realpath(). 回调函数有两个参数 err, resolvedPath。
fs.realpathSync(path[, cache])	同步 realpath()。返回绝对路径。
fs.unlink(path, callback)	异步 unlink().回调函数没有参数，但可能抛出异常。
fs.unlinkSync(path)	同步 unlink().
fs.rmdir(path, callback)	异步 rmdir().回调函数没有参数，但可能抛出异常。
fs.rmdirSync(path)	同步 rmdir().
fs.mkdir(path[, mode], callback)	S异步 mkdir(2).回调函数没有参数，但可能抛出异常。 mode defaults to 0777.
fs.mkdirSync(path[, mode])	同步 mkdir().
fs.readdir(path, callback)	异步 readdir(3). 读取目录的内容。
fs.readdirSync(path)	同步 readdir().返回文件数组列表。
fs.close(fd, callback)	异步 close().回调函数没有参数，但可能抛出异常。
fs.closeSync(fd)	同步 close().
fs.open(path, flags[, mode], callback)	异步打开文件。
fs.openSync(path, flags[, mode])	同步 version of fs.open().
fs.utimes(path, atime, mtime, callback)	?
fs.utimesSync(path, atime, mtime)	修改文件时间戳，文件通过指定的文件路径。
fs.futimes(fd, atime, mtime, callback)	?
fs.futimesSync(fd, atime, mtime)	修改文件时间戳，通过文件描述符指定。
fs.fsync(fd, callback)	异步 fsync.回调函数没有参数，但可能抛出异常。
fs.fsyncSync(fd)	同步 fsync.
fs.write(fd, buffer, offset, length[, position], callback)	将缓冲区内容写入到通过文件描述符指定的文件。
fs.write(fd, data[, position[,	

encoding]], callback)	
fs.writeSync(fd, buffer, offset, length[, position])	同步版的 fs.write()。
fs.writeSync(fd, data[, position[, encoding]])	同步版的 fs.write().
fs.read(fd, buffer, offset, length, position, callback)	通过文件描述符 fd 读取文件内容。
fs.readSync(fd, buffer, offset, length, position)	同步版的 fs.read.
fs.readFile(filename[, options], callback)	异步读取文件内容。
fs.readFileSync(filename[, options])	
fs.writeFile(filename, data[, options], callback)	异步写入文件内容。
fs.writeFileSync(filename, data[, options])	同步版的 fs.writeFile。
fs.appendFile(filename, data[, options], callback)	异步追加文件内容。
fs.appendFileSync(filename, data[, options])	The 同步 version of fs.appendFile.
fs.watchFile(filename[, options], listener)	查看文件的修改。
fs.unwatchFile(filename[, listener])	停止查看 filename 的修改。
fs.watch(filename[, options][, listener])	查看 filename 的修改，filename 可以是文件或目录。返回 fs.FSWatcher 对象。
fs.exists(path, callback)	检测给定的路径是否存在。
fs.existsSync(path)	同步版的 fs.exists.
fs.access(path[, mode], callback)	测试指定路径用户权限。
fs.accessSync(path[, mode])	同步版的 fs.access。
fs.createReadStream(path[, options])	返回ReadStream 对象。
fs.createWriteStream(path[, options])	返回 WriteStream 对象。
fs.symlink(srcpath, dstpath[, type], callback)	异步 symlink().回调函数没有参数，但可能抛出异常。

dstpath[, type], callback)	

更多内容，请查看官网文件模块描述：File System。

Node.js GET/POST请求

在很多场景中，我们的服务器都需要跟用户的浏览器打交道，如表单提交。

表单提交到服务器一般都使用GET/POST请求。

本章节我们将为大家介绍 Node.js GET/POST请求。

获取GET请求内容

由于GET请求直接被嵌入在路径中，URL是完整的请求路径，包括了?后面的部分，因此你可以手动解析后面的内容作为GET请求的参数。

node.js中url模块中的parse函数提供了这个功能。

```
var http = require('http');
var url = require('url');
var util = require('util');

http.createServer(function(req, res){
    res.writeHead(200, {'Content-Type': 'text/plain'});
    res.end(util.inspect(url.parse(req.url, true)));
}).listen(3000);
```

在浏览器中访问 `http://localhost:3000/user?name=w3c&email=w3c@w3cschool.cc` 然后查看返回结果:

获取POST请求内容

POST请求的内容全部的都在请求体中，http.ServerRequest并没有一个属性内容为请求体，原因是等待请求体传输可能是一件耗时的工作。

比如上传文件，而很多时候我们可能并不需要理会请求体的内容，恶意的POST请求会大大消耗服务器的资源，所有node.js默认是不会解析请求体的， 当你需要的时候，需要手动来做。

```
var http = require('http');
var querystring = require('querystring');
var util = require('util');

http.createServer(function(req, res){
    var post = '';     //定义了一个post变量，用于暂存请求体的信息

    req.on('data', function(chunk){    //通过req的data事件监听函数，每当接受到请求体的数据，就累加
        post += chunk;
    });

    req.on('end', function(){    //在end事件触发后，通过querystring.parse将post解析为真正的POST
        post = querystring.parse(post);
        res.end(util.inspect(post));
    });
}).listen(3000);
```

Node.js 工具模块

在 Node.js 模块库中有很多好用的模块。接下来我们为大家介绍几种常用模块的使用：

模块名	描述
OS 模块	提供基本的系统操作函数。
Path 模块	提供了处理和转换文件路的工具。
Net 模块	用于底层的网络通信。提供了服务端和客户端的的操作。
DNS 模块	用于解析域名。
Domain 模块	简化异步代码的异常处理，可以捕捉处理try catch无法捕捉的。

Node.js OS 模块

Node.js os 模块提供了一些基本的系统操作函数。我们可以通过以下方式引入该模块：

```
var os =  require("os")
```

方法

方法	描述
os.tmpdir()	返回操作系统的默认临时文件夹。
os.endianness()	返回 CPU 的字节序，可能的是 "BE" 或 "LE"。
os.hostname()	返回操作系统的主机名。
os.type()	返回操作系统名
os.platform()	返回操作系统名
os.arch()	返回操作系统 CPU 架构，可能的值有 "x64"、"arm" 和 "ia32"。
os.release()	返回操作系统的发行版本。
os.uptime()	返回操作系统运行的时间，以秒为单位。
os.loadavg()	返回一个包含 1、5、15 分钟平均负载的数组。
os.totalmem()	返回系统内存总量，单位为字节。
os.freemem()	返回操作系统空闲内存量，单位是字节。
os.cpus()	返回一个对象数组，包含所安装的每个 CPU/内核的信息：型号、速度（单位 MHz）、时间（一个包含 user、nice、sys、idle 和 irq 所使用 CPU/内核毫秒数的对象）。
os.networkInterfaces()	获得网络接口列表。

属性

属性	描述
os.EOL	定义了操作系统的行尾符的常量。

实例

创建 main.js 文件，代码如下所示：

```
var os =  require("os");  // CPU 的字节序 console.log('endianness : '  + os.endianness());
```

代码执行结果如下：

```
$ node main.js
endianness : LE
type :  Linux platform : linux
total memory :  25103400960 bytes. free memory :  20676710400 bytes.
```

Node.js Path 模块

Node.js path 模块提供了一些用于处理文件路径的小工具，我们可以通过以下方式引入该模块：

```
var path = require("path")
```

方法

方法	描述
path.normalize(p)	规范化路径，注意'..' 和 '.'。
path.join([path1][, path2][, ...])	用于连接路径。该方法的主要用途在于，会正确使用当前系统的路径分隔符，Unix系统是"/"，Windows系统是"\"。
path.resolve([from ...], to)	将 **to** 参数解析为绝对路径。
path.isAbsolute(path)	判断参数 **path** 是否是绝对路径。
path.relative(from, to)	用于将相对路径转为绝对路径。
path.dirname(p)	返回路径中代表文件夹的部分，同 Unix 的dirname 命令类似。
path.basename(p[, ext])	返回路径中的最后一部分。同 Unix 命令 bashname 类似。
path.extname(p)	返回路径中文件的后缀名，即路径中最后一个'.'之后的部分。如果一个路径中并不包含'.'或该路径只包含一个'.' 且这个'.'为路径的第一个字符，则此命令返回空字符串。
path.parse(pathString)	返回路径字符串的对象。
path.format(pathObject)	从对象中返回路径字符串，和 path.parse 相反。

属性

属性	描述
path.sep	平台的文件路径分隔符，'\' 或 '/'。
path.delimiter	平台的分隔符, ; or ':'.
path.posix	提供上述 path 的方法，不过总是以 posix 兼容的方式交互。
path.win32	提供上述 path 的方法，不过总是以 win32 兼容的方式交互。

实例

创建 main.js 文件，代码如下所示：

```
var path = require("path");

// 格式化路径
console.log('normalization : ' + path.normalize('/test/test1//2slashes/1slash/tab/..'));

// 连接路径
console.log('joint path : ' + path.join('/test', 'test1', '2slashes/1slash', 'tab', '..')

// 转换为绝对路径
console.log('resolve : ' + path.resolve('main.js'));

// 路径中文件的后缀名
console.log('ext name : ' + path.extname('main.js'));
```

代码执行结果如下：

```
$ node main.js
normalization : /test/test1/2slashes/1slash
joint path : /test/test1/2slashes/1slash
resolve : /web/com/1427176256_27423/main.js
ext name : .js
```

Node.js Net 模块

Node.js Net 模块提供了一些用于底层的网络通信的小工具，包含了创建服务器/客户端的方法，我们可以通过以下方式引入该模块：

```
var net = require("net")
```

方法

方法	描述
net.createServer([options][, connectionListener])	创建一个 TCP 服务器。参数 connectionListener 自动给 'connection' 事件创建监听器。
net.connect(options[, connectionListener])	返回一个新的 'net.Socket'，并连接到指定的地址和端口。 当 socket 建立的时候，将会触发 'connect' 事件。
net.createConnection(options[, connectionListener])	创建一个到端口 port 和 主机 host的 TCP 连接。 host 默认为 'localhost'。
net.connect(port[, host][, connectListener])	创建一个端口为 port 和主机为 host的 TCP 连接 。host 默认为 'localhost'。参数 connectListener 将会作为监听器添加到 'connect' 事件。返回 'net.Socket'。
net.createConnection(port[, host][, connectListener])	创建一个端口为 port 和主机为 host的 TCP 连接 。host 默认为 'localhost'。参数 connectListener 将会作为监听器添加到 'connect' 事件。返回 'net.Socket'。
net.connect(path[, connectListener])	创建连接到 path 的 unix socket 。参数 connectListener 将会作为监听器添加到 'connect' 事件上。返回 'net.Socket'。
net.createConnection(path[, connectListener])	创建连接到 path 的 unix socket 。参数 connectListener 将会作为监听器添加到 'connect' 事件。返回 'net.Socket'。
net.isIP(input)	检测输入的是否为 IP 地址。 IPV4 返回 4， IPV6 返回 6，其他情况返回 0。
net.isIPv4(input)	如果输入的地址为 IPV4， 返回 true，否则返回 false。
net.isIPv6(input)	如果输入的地址为 IPV6， 返回 true，否则返回 false。

net.Server

net.Server通常用于创建一个 TCP 或本地服务器。

方法	描述
server.listen(port[, host][, backlog][, callback])	监听指定端口 port 和 主机 host ac连接。 默认情况下 host 接受任何 IPv4 地址(INADDR_ANY)的直接连接。端口 port 为 0 时，则会分配一个随机端口。
server.listen(path[, callback])	通过指定 path 的连接，启动一个本地 socket 服务器。
server.listen(handle[, callback])	通过指定句柄连接。
server.listen(options[, callback])	options 的属性：端口 port, 主机 host, 和 backlog, 以及可选参数 callback 函数, 他们在一起调用 server.listen(port, [host], [backlog], [callback])。还有，参数 path 可以用来指定 UNIX socket。
server.close([callback])	服务器停止接收新的连接，保持现有连接。这是异步函数，当所有连接结束的时候服务器会关闭，并会触发 'close' 事件。
server.address()	操作系统返回绑定的地址，协议族名和服务器端口。
server.unref()	如果这是事件系统中唯一一个活动的服务器，调用 unref 将允许程序退出。
server.ref()	与 unref 相反，如果这是唯一的服务器，在之前被 unref 了的服务器上调用 ref 将不会让程序退出（默认行为）。如果服务器已经被 ref，则再次调用 ref 并不会产生影响。
server.getConnections(callback)	异步获取服务器当前活跃连接的数量。当 socket 发送给子进程后才有效；回调函数有 2 个参数 err 和 count。

事件

事件	描述
listening	当服务器调用 server.listen 绑定后会触发。
connection	当新连接创建后会被触发。socket 是 net.Socket实例。
close	服务器关闭时会触发。注意，如果存在连接，这个事件不会被触发直到所有的连接关闭。
error	发生错误时触发。'close' 事件将被下列事件直接调用。

net.Socket

net.Socket 对象是 TCP 或 UNIX Socket 的抽象。net.Socket 实例实现了一个双工流接口。他们可以在用户创建客户端(使用 connect())时使用, 或者由 Node 创建它们，并通过 connection 服务器事件传递给用户。

事件

net.Socket 事件有：

事件	描述
lookup	在解析域名后，但在连接前，触发这个事件。对 UNIX sokcet 不适用。
connect	成功建立 socket 连接时触发。
data	当接收到数据时触发。
end	当 socket 另一端发送 FIN 包时，触发该事件。
timeout	当 socket 空闲超时时触发，仅是表明 socket 已经空闲。用户必须手动关闭连接。
drain	当写缓存为空得时候触发。可用来控制上传。
error	错误发生时触发。
close	当 socket 完全关闭时触发。参数 had_error 是布尔值，它表示是否因为传输错误导致 socket 关闭。

属性

net.Socket 提供了很多有用的属性，便于控制 socket 交互：

属性	描述
socket.bufferSize	该属性显示了要写入缓冲区的字节数。
socket.remoteAddress	远程的 IP 地址字符串，例如：'74.125.127.100' or '2001:4860:a005::68'。
socket.remoteFamily	远程IP协议族字符串，比如 'IPv4' or 'IPv6'。
socket.remotePort	远程端口，数字表示，例如：80 or 21。
socket.localAddress	网络连接绑定的本地接口 远程客户端正在连接的本地 IP 地址，字符串表示。例如，如果你在监听'0.0.0.0'而客户端连接在'192.168.1.1'，这个值就会是 '192.168.1.1'。
socket.localPort	本地端口地址，数字表示。例如：80 or 21。
socket.bytesRead	接收到得字节数。
socket.bytesWritten	发送的字节数。

方法

方法	描述
new net.Socket([options])	构造一个新的 socket 对象。
socket.connect(port[, host][, connectListener])	指定端口 port 和 主机 host，创建 socket 连接 。参数 host 默认为 localhost。通常情况不需要使用 net.createConnection 打开 socket。只有你实现了自己的 socket 时才会用到。
socket.connect(path[, connectListener])	打开指定路径的 unix socket。通常情况不需要使用 net.createConnection 打开 socket。只有你实现了自己的 socket 时才会用到。
socket.setEncoding([encoding])	设置编码
socket.write(data[, encoding][, callback])	在 socket 上发送数据。第二个参数指定了字符串的编码，默认是 UTF8 编码。
socket.end([data][, encoding])	半关闭 socket。例如，它发送一个 FIN 包。可能服务器仍在发送数据。
socket.destroy()	确保没有 I/O 活动在这个套接字上。只有在错误发生情况下才需要。（处理错误等等）。
socket.pause()	暂停读取数据。就是说，不会再触发 data 事件。对于控制上传非常有用。
socket.resume()	调用 pause() 后想恢复读取数据。
socket.setTimeout(timeout[, callback])	socket 闲置时间超过 timeout 毫秒后 ，将 socket 设置为超时。
socket.setNoDelay([noDelay])	禁用纳格（Nagle）算法。默认情况下 TCP 连接使用纳格算法，在发送前他们会缓冲数据。将 noDelay 设置为 true 将会在调用 socket.write() 时立即发送数据。noDelay 默认值为 true。
socket.setKeepAlive([enable][, initialDelay])	禁用/启用长连接功能，并在发送第一个在闲置 socket 上的长连接 probe 之前，可选地设定初始延时。默认为 false。 设定 initialDelay （毫秒），来设定收到的最后一个数据包和第一个长连接 probe之间的延时。将 initialDelay 设为0，将会保留默认（或者之前）的值。默认值为0.
socket.address()	操作系统返回绑定的地址，协议族名和服务器端口。返回的对象有 3 个属性，比如{ port: 12346, family: 'IPv4', address: '127.0.0.1' }。
socket.unref()	如果这是事件系统中唯一一个活动的服务器，调用 unref 将允许程序退出。如果服务器已被 unref，则再次调用 unref 并不会产生影响。
socket.ref()	与 unref 相反，如果这是唯一的服务器，在之前被 unref 了的服务器上调用 ref 将不会让程序退出（默认行为）。如果服务器已经被 ref，则再次调用 ref 并不会产生影响。

实例

创建 server.js 文件，代码如下所示：

```
var net = require('net');
var server = net.createServer(function(connection) {
   console.log('client connected');
   connection.on('end', function() {
      console.log('客户端关闭连接');
   });
   connection.write('Hello World!\r\n');
   connection.pipe(connection);
});
server.listen(8080, function() {
  console.log('server is listening');
});
```

执行以上服务端代码：

```
$ node server.js
server is listening   # 服务已创建并监听 8080 端口
```

新开一个窗口，创建 client.js 文件，代码如下所示：

```
var net = require('net');
var client = net.connect({port: 8080}, function() {
   console.log('连接到服务器！');
});
client.on('data', function(data) {
   console.log(data.toString());
   client.end();
});
client.on('end', function() {
   console.log('断开与服务器的连接');
});
```

执行以上客户端的代码：

```
连接到服务器！
Hello World!

断开与服务器的连接
```

Gif 实例演示

Node.js DNS 模块

Node.js **DNS** 模块用于解析域名。引入 DNS 模块语法格式如下：

```
var dns = require("dns")
```

方法

方法	描述
dns.lookup(hostname[, options], callback)	将域名（比如 'runoob.com'）解析为第一条找到的记录 A （IPV4）或 AAAA(IPV6)。参数 options可以是一个对象或整数。如果没有提供 options，IP v4 和 v6 地址都可以。如果 options 是整数，则必须是 4 或 6。
dns.lookupService(address, port, callback)	使用 getnameinfo 解析传入的地址和端口为域名和服务。
dns.resolve(hostname[, rrtype], callback)	将一个域名（如 'runoob.com'）解析为一个 rrtype 指定记录类型的数组。
dns.resolve4(hostname, callback)	和 dns.resolve() 类似, 仅能查询 IPv4 (A 记录）。 addresses IPv4 地址数组 (比如，['74.125.79.104', '74.125.79.105', '74.125.79.106']）。
dns.resolve6(hostname, callback)	和 dns.resolve4() 类似， 仅能查询 IPv6(AAAA 查询）
dns.resolveMx(hostname, callback)	和 dns.resolve() 类似, 仅能查询邮件交换(MX 记录)。
dns.resolveTxt(hostname, callback)	和 dns.resolve() 类似, 仅能进行文本查询 (TXT 记录）。 addresses 是 2-d 文本记录数组。(比如，[['v=spf1 ip4:0.0.0.0 ', '~all']]）。 每个子数组包含一条记录的 TXT 块。根据使用情况可以连接在一起，也可单独使用。
dns.resolveSrv(hostname, callback)	和 dns.resolve() 类似, 仅能进行服务记录查询 (SRV 记录）。 addresses 是 hostname可用的 SRV 记录数组。 SRV 记录属性有优先级（priority），权重（weight）, 端口（port）, 和名字（name） (比如，[{'priority': 10, 'weight': 5, 'port': 21223, 'name': 'service.example.com'}, ...]）。
dns.resolveSoa(hostname, callback)	和 dns.resolve() 类似, 仅能查询权威记录(SOA 记录）。
dns.resolveNs(hostname, callback)	和 dns.resolve() 类似, 仅能进行域名服务器记录查询(NS 记录）。 addresses 是域名服务器记录数组（hostname 可以使用） (比如, ['ns1.example.com', 'ns2.example.com']）。
dns.resolveCname(hostname, callback)	和 dns.resolve() 类似, 仅能进行别名记录查询 (CNAME记录)。addresses 是对 hostname 可用的别名记录数组 (比如，, ['bar.example.com']）。
dns.reverse(ip, callback)	反向解析 IP 地址，指向该 IP 地址的域名数组。
dns.getServers()	返回一个用于当前解析的 IP 地址数组的字符串。
dns.setServers(servers)	指定一组 IP 地址作为解析服务器。

rrtypes

以下列出了 dns.resolve() 方法中有效的 rrtypes值:

- `'A'` IPV4 地址, 默认
- `'AAAA'` IPV6 地址
- `'MX'` 邮件交换记录
- `'TXT'` text 记录
- `'SRV'` SRV 记录
- `'PTR'` 用来反向 IP 查找
- `'NS'` 域名服务器记录
- `'CNAME'` 别名记录
- `'SOA'` 授权记录的初始值

错误码

每次 DNS 查询都可能返回以下错误码：

- `dns.NODATA`：无数据响应。
- `dns.FORMERR`：查询格式错误。
- `dns.SERVFAIL`：常规失败。
- `dns.NOTFOUND`：没有找到域名。
- `dns.NOTIMP`：未实现请求的操作。
- `dns.REFUSED`：拒绝查询。
- `dns.BADQUERY`：查询格式错误。
- `dns.BADNAME`：域名格式错误。
- `dns.BADFAMILY`：地址协议不支持。
- `dns.BADRESP`：回复格式错误。
- `dns.CONNREFUSED`：无法连接到 DNS 服务器。
- `dns.TIMEOUT`：连接 DNS 服务器超时。
- `dns.EOF`：文件末端。
- `dns.FILE`：读文件错误。
- `dns.NOMEM`：内存溢出。
- `dns.DESTRUCTION`：通道被摧毁。
- `dns.BADSTR`：字符串格式错误。
- `dns.BADFLAGS`：非法标识符。
- `dns.NONAME`：所给主机不是数字。
- `dns.BADHINTS`：非法HINTS标识符。
- `dns.NOTINITIALIZED`：c c-ares 库尚未初始化。
- `dns.LOADIPHLPAPI`：加载 iphlpapi.dll 出错。
- `dns.ADDRGETNETWORKPARAMS`：无法找到 GetNetworkParams 函数。
- `dns.CANCELLED`：取消 DNS 查询。

实例

创建 main.js 文件，代码如下所示：

```
var dns = require('dns');

dns.lookup('www.github.com', function onLookup(err, address, family) {
   console.log('ip 地址:', address);
   dns.reverse(address, function (err, hostnames) {
   if (err) {
      console.log(err.stack);
   }

   console.log('反向解析 ' + address + ': ' + JSON.stringify(hostnames));
});
});
```

执行以上代码，结果如下所示:

```
address: 192.30.252.130
reverse for 192.30.252.130: ["github.com"]
```

Node.js Domain 模块

Node.js **Domain(域)** 简化异步代码的异常处理，可以捕捉处理try catch无法捕捉的异常。引入 Domain 模块 语法格式如下：

```
var domain = require("domain")
```

domain模块，把处理多个不同的IO的操作作为一个组。注册事件和回调到domain，当发生一个错误事件或抛出一个错误时，domain对象会被通知，不会丢失上下文环境，也不导致程序错误立即推出，与process.on('uncaughtException')不同。

Domain 模块可分为隐式绑定和显式绑定：

- 隐式绑定: 把在domain上下文中定义的变量，自动绑定到domain对象
- 显式绑定: 把不是在domain上下文中定义的变量，以代码的方式绑定到domain对象

方法

方法	描述
domain.run(function)	在域的上下文运行提供的函数，隐式的绑定了所有的事件分发器，计时器和底层请求。
domain.add(emitter)	显式的增加事件
domain.remove(emitter)	删除事件。
domain.bind(callback)	返回的函数是一个对于所提供的回调函数的包装函数。当调用这个返回的函数被时，所有被抛出的错误都会被导向到这个域的 error 事件。
domain.intercept(callback)	和 domain.bind(callback) 类似。除了捕捉被抛出的错误外，它还会拦截 Error 对象作为参数传递到这个函数。
domain.enter()	进入一个异步调用的上下文，绑定到domain。
domain.exit()	退出当前的domain，切换到不同的链的异步调用的上下文中。对应domain.enter()。
domain.dispose()	释放一个domain对象，让node进程回收这部分资源。
domain.create()	返回一个domain对象。

属性

属性	描述
domain.members	已加入domain对象的域定时器和事件发射器的数组。

实例

创建 main.js 文件，代码如下所示：

```
var EventEmitter = require("events").EventEmitter;
var domain = require("domain");

var emitter1 = new EventEmitter();

// 创建域
var domain1 = domain.create();

domain1.on('error', function(err){
   console.log("domain1 处理这个错误 ("+err.message+")");
});

// 显式绑定
domain1.add(emitter1);

emitter1.on('error',function(err){
   console.log("监听器处理此错误 ("+err.message+")");
});

emitter1.emit('error',new Error('通过监听器来处理'));

emitter1.removeAllListeners('error');

emitter1.emit('error',new Error('通过 domain1 处理'));

var domain2 = domain.create();

domain2.on('error', function(err){
   console.log("domain2 处理这个错误 ("+err.message+")");
});

// 隐式绑定
domain2.run(function(){
   var emitter2 = new EventEmitter();
   emitter2.emit('error',new Error('通过 domain2 处理'));
});

domain1.remove(emitter1);
emitter1.emit('error', new Error('转换为异常，系统将崩溃!'));
```

执行以上代码，结果如下所示:

```
监听器处理此错误 (通过监听器来处理)
domain1 处理这个错误 (通过 domain1 处理)
domain2 处理这个错误 (通过 domain2 处理)

events.js:72
        throw er; // Unhandled 'error' event
              ^
Error: 转换为异常，系统将崩溃！
    at Object.<anonymous> (/www/node/main.js:40:24)
    at Module._compile (module.js:456:26)
    at Object.Module._extensions..js (module.js:474:10)
    at Module.load (module.js:356:32)
    at Function.Module._load (module.js:312:12)
    at Function.Module.runMain (module.js:497:10)
    at startup (node.js:119:16)
    at node.js:929:3
```

Node.js Web 模块

什么是 Web 服务器？

Web服务器一般指网站服务器，是指驻留于因特网上某种类型计算机的程序，Web服务器的基本功能就是提供Web信息浏览服务。它只需支持HTTP协议、HTML文档格式及URL，与客户端的网络浏览器配合。

大多数 web 服务器都支持服务端的脚本语言（php、python、ruby）等，并通过脚本语言从数据库获取数据，将结果返回给客户端浏览器。

目前最主流的三个Web服务器是Apache、Nginx、IIS。

Web 应用架构

- **Client** - 客户端，一般指浏览器，浏览器可以通过 HTTP 协议向服务器请求数据。
- **Server** - 服务端，一般指 Web 服务器，可以接收客户端请求，并向客户端发送响应数据。
- **Business** - 业务层， 通过 Web 服务器处理应用程序，如与数据库交互，逻辑运算，调用外部程序等。
- **Data** - 数据层，一般由数据库组成。

使用 Node 创建 Web 服务器

Node.js 提供了 http 模块，http 模块主要用于搭建 HTTP 服务端和客户端，使用 HTTP 服务器或客户端功能必须调用 http 模块，代码如下：

```
var http = require('http');
```

以下是演示一个最基本的 HTTP 服务器架构(使用8081端口)，创建 server.js 文件，代码如下所示：

```
var http = require('http');
var fs = require('fs');
var url = require('url');

// 创建服务器
http.createServer( function (request, response) {
   // 解析请求，包括文件名
   var pathname = url.parse(request.url).pathname;

   // 输出请求的文件名
   console.log("Request for " + pathname + " received.");

   // 从文件系统中读取请求的文件内容
   fs.readFile(pathname.substr(1), function (err, data) {
      if (err) {
         console.log(err);
         // HTTP 状态码: 404 : NOT FOUND
         // Content Type: text/plain
         response.writeHead(404, {'Content-Type': 'text/html'});
      }else{
         // HTTP 状态码: 200 : OK
         // Content Type: text/plain
         response.writeHead(200, {'Content-Type': 'text/html'});

         // 响应文件内容
         response.write(data.toString());
      }
      //  发送响应数据
      response.end();
   });
}).listen(8081);

// 控制台会输出以下信息
console.log('Server running at http://127.0.0.1:8081/');
```

接下来我们在该目录下创建一个 index.htm 文件，代码如下：

```
<html>
<head>
<title>Sample Page</title>
</head>
<body>
Hello World!
</body>
</html>
```

执行 server.js 文件：

```
$ node server.js
Server running at http://127.0.0.1:8081/
```

接着我们在浏览器中打开地址：http://127.0.0.1:8081/index.htm，显示如下图所示:

执行 server.js 的控制台输出信息如下：

```
Server running at http://127.0.0.1:8081/
Request for /index.htm received.     #  客户端请求信息
```

Gif 实例演示

使用 Node 创建 Web 客户端

Node 创建 Web 客户端需要引入 http 模块，创建 client.js 文件，代码如下所示：

```
<pre>
var http = require('http');

// 用于请求的选项
var options = {
   host: 'localhost',
   port: '8081',
   path: '/index.htm'
};

// 处理响应的回调函数
var callback = function(response){
   // 不断更新数据
   var body = '';
   response.on('data', function(data) {
      body += data;
   });

   response.on('end', function() {
      // 数据接收完成
      console.log(body);
   });
}
// 向服务端发送请求
var req = http.request(options, callback);
req.end();
```

新开一个终端，执行 client.js 文件，输出结果如下：

```
$ node client.js
<html>
<head>
<title>Sample Page</title>
</head>
<body>
Hello World!
</body>
</html>
```

执行 server.js 的控制台输出信息如下：

```
Server running at http://127.0.0.1:8081/
Request for /index.htm received.   # 客户端请求信息
```

Gif 实例演示

Node.js Express 框架

Express 简介

Express 是一个简洁而灵活的 node.js Web应用框架, 提供了一系列强大特性帮助你创建各种 Web 应用，和丰富的 HTTP 工具。

使用 Express 可以快速地搭建一个完整功能的网站。

Express 框架核心特性：

- 可以设置中间件来响应 HTTP 请求。
- 定义了路由表用于执行不同的 HTTP 请求动作。
- 可以通过向模板传递参数来动态渲染 HTML 页面。

安装 Express

安装 Express 并将其保存到依赖列表中：

```
$ npm install express --save
```

以上命令会将 Express 框架安装在当期目录的 **node_modules** 目录中， **node_modules** 目录下会自动创建 express 目录。以下几个重要的模块是需要与 express 框架一起安装的：

- **body-parser** - node.js 中间件，用于处理 JSON, Raw, Text 和 URL 编码的数据。
- **cookie-parser** - 这就是一个解析Cookie的工具。通过req.cookies可以取到传过来的cookie，并把它们转成对象。
- **multer** - node.js 中间件，用于处理 enctype="multipart/form-data"（设置表单的MIME编码）的表单数据。

```
$ npm install body-parser --save
$ npm install cookie-parser --save
$ npm install multer --save
```

第一个 Express 框架实例

接下来我们使用 Express 框架来输出 "Hello World"。

以下实例中我们引入了 express 模块，并在客户端发起请求后，响应 "Hello World" 字符串。

创建 express_demo.js 文件，代码如下所示：

```
//express_demo.js 文件
var express = require('express');
var app = express();

app.get('/', function (req, res) {
   res.send('Hello World');
})

var server = app.listen(8081, function () {

  var host = server.address().address
  var port = server.address().port

  console.log("应用实例，访问地址为 http://%s:%s", host, port)

})
```

执行以上代码：

```
$ node express_demo.js
应用实例，访问地址为 http://0.0.0.0:8081
```

在浏览器中访问 http://127.0.0.1:8081，结果如下图所示：

请求和响应

Express 应用使用回调函数的参数： **request** 和 **response** 对象来处理请求和响应的数据。

```
app.get('/', function (req, res) {
   // --
})
```

request 和 **response** 对象的具体介绍：

Request 对象 - request 对象表示 HTTP 请求，包含了请求查询字符串，参数，内容，HTTP 头部等属性。常见属性有：

1. req.app：当callback为外部文件时，用req.app访问express的实例
2. req.baseUrl：获取路由当前安装的URL路径
3. req.body / req.cookies：获得「请求主体」/ Cookies
4. req.fresh / req.stale：判断请求是否还「新鲜」
5. req.hostname / req.ip：获取主机名和IP地址
6. req.originalUrl：获取原始请求URL
7. req.params：获取路由的parameters

8. req.path：获取请求路径
9. req.protocol：获取协议类型
10. req.query：获取URL的查询参数串
11. req.route：获取当前匹配的路由
12. req.subdomains：获取子域名
13. req.accpets（）：检查请求的Accept头的请求类型
14. req.acceptsCharsets / req.acceptsEncodings / req.acceptsLanguages
15. req.get（）：获取指定的HTTP请求头
16. req.is（）：判断请求头Content-Type的MIME类型

Response 对象 - response 对象表示 HTTP 响应，即在接收到请求时向客户端发送的 HTTP 响应数据。常见属性有：

1. res.app：同req.app一样
2. res.append（）：追加指定HTTP头
3. res.set（）在res.append（）后将重置之前设置的头
4. res.cookie（name，value [，option]）：设置Cookie
5. opition: domain / expires / httpOnly / maxAge / path / secure / signed
6. res.clearCookie（）：清除Cookie
7. res.download（）：传送指定路径的文件
8. res.get（）：返回指定的HTTP头
9. res.json（）：传送JSON响应
10. res.jsonp（）：传送JSONP响应
11. res.location（）：只设置响应的Location HTTP头，不设置状态码或者close response
12. res.redirect（）：设置响应的Location HTTP头，并且设置状态码302
13. res.send（）：传送HTTP响应
14. res.sendFile（path [，options] [，fn]）：传送指定路径的文件 -会自动根据文件extension设定Content-Type
15. res.set（）：设置HTTP头，传入object可以一次设置多个头
16. res.status（）：设置HTTP状态码
17. res.type（）：设置Content-Type的MIME类型

路由

我们已经了解了 HTTP 请求的基本应用，而路由决定了由谁(指定脚本)去响应客户端请求。

在HTTP请求中，我们可以通过路由提取出请求的URL以及GET/POST参数。

接下来我们扩展 Hello World，添加一些功能来处理更多类型的 HTTP 请求。

创建 express_demo2.js 文件，代码如下所示：

```
var express = require('express');
var app = express();

//  主页输出 "Hello World"
app.get('/', function (req, res) {
   console.log("主页 GET 请求");
   res.send('Hello GET');
})

//  POST 请求
app.post('/', function (req, res) {
   console.log("主页 POST 请求");
   res.send('Hello POST');
})

//  /del_user 页面响应
app.delete('/del_user', function (req, res) {
   console.log("/del_user 响应 DELETE 请求");
   res.send('删除页面');
})

//  /list_user 页面 GET 请求
app.get('/list_user', function (req, res) {
   console.log("/list_user GET 请求");
   res.send('用户列表页面');
})

// 对页面 abcd, abxcd, ab123cd, 等响应 GET 请求
app.get('/ab*cd', function(req, res) {
   console.log("/ab*cd GET 请求");
   res.send('正则匹配');
})

var server = app.listen(8081, function () {

  var host = server.address().address
  var port = server.address().port

  console.log("应用实例，访问地址为 http://%s:%s", host, port)

})
```

执行以上代码：

```
$ node express_demo2.js
应用实例，访问地址为 http://0.0.0.0:8081
```

接下来你可以尝试访问 http://127.0.0.1:8081 不同的地址，查看效果。

在浏览器中访问 http://127.0.0.1:8081/list_user，结果如下图所示：

在浏览器中访问 http://127.0.0.1:8081/abcd，结果如下图所示：

在浏览器中访问 http://127.0.0.1:8081/abcdefg，结果如下图所示：

静态文件

Express 提供了内置的中间件 **express.static** 来设置静态文件如：图片， CSS, JavaScript 等。

你可以使用 **express.static** 中间件来设置静态文件路径。例如，如果你将图片， CSS, JavaScript 文件放在 public 目录下，你可以这么写：

```
app.use(express.static('public'));
```

我们可以到 public/images 目录下放些图片,如下所示：

```
node_modules
server.js
public/
public/images
public/images/logo.png
```

让我们再修改下 "Hello Word" 应用添加处理静态文件的功能。

创建 express_demo3.js 文件，代码如下所示：

```
var express = require('express');
var app = express();

app.use(express.static('public'));

app.get('/', function (req, res) {
   res.send('Hello World');
})

var server = app.listen(8081, function () {

  var host = server.address().address
  var port = server.address().port

  console.log("应用实例，访问地址为 http://%s:%s", host, port)

})
```

执行以上代码：

```
$ node express_demo3.js
应用实例，访问地址为 http://0.0.0.0:8081
```

执行以上代码：

在浏览器中访问 http://127.0.0.1:8081/images/logo.png（本实例采用了菜鸟教程的logo），结果如下图所示：

GET 方法

以下实例演示了在表单中通过 GET 方法提交两个参数，我们可以使用 server.js 文件内的 **process_get** 路由器来处理输入：

index.htm 文件代码如下：

```
<html>
<body>
<form action="http://127.0.0.1:8081/process_get" method="GET">
First Name: <input type="text" name="first_name">  <br>

Last Name: <input type="text" name="last_name">
<input type="submit" value="Submit">
</form>
</body>
</html>
```

server.js 文件代码如下:

```
var express = require('express');
var app = express();

app.use(express.static('public'));

app.get('/index.htm', function (req, res) {
   res.sendFile( __dirname + "/" + "index.htm" );
})

app.get('/process_get', function (req, res) {

   // 输出 JSON 格式
   response = {
       first_name:req.query.first_name,
       last_name:req.query.last_name
   };
   console.log(response);
   res.end(JSON.stringify(response));
})

var server = app.listen(8081, function () {

  var host = server.address().address
  var port = server.address().port

  console.log("应用实例，访问地址为 http://%s:%s", host, port)

})
```

执行以上代码：

```
node server.js
应用实例，访问地址为 http://0.0.0.0:8081
```

浏览器访问 http://127.0.0.1:8081/index.htm，如图所示：

现在你可以向表单输入数据，并提交，如下演示：

POST 方法

以下实例演示了在表单中通过 POST 方法提交两个参数，我们可以使用 server.js 文件内的 **process_get** 路由器来处理输入：

index.htm 文件代码修改如下：

```
<html>
<body>
<form action="http://127.0.0.1:8081/process_post" method="POST">
First Name: <input type="text" name="first_name">  <br>

Last Name: <input type="text" name="last_name">
<input type="submit" value="Submit">
</form>
</body>
</html>
```

server.js 文件代码修改如下:

```
var express = require('express');
var app = express();
var bodyParser = require('body-parser');

// 创建 application/x-www-form-urlencoded 编码解析
var urlencodedParser = bodyParser.urlencoded({ extended: false })

app.use(express.static('public'));

app.get('/index.htm', function (req, res) {
   res.sendFile( __dirname + "/" + "index.htm" );
})

app.post('/process_post', urlencodedParser, function (req, res) {

   // 输出 JSON 格式
   response = {
       first_name:req.body.first_name,
       last_name:req.body.last_name
   };
   console.log(response);
   res.end(JSON.stringify(response));
})

var server = app.listen(8081, function () {

  var host = server.address().address
  var port = server.address().port

  console.log("应用实例，访问地址为 http://%s:%s", host, port)

})
```

执行以上代码：

```
$ node express_demo.js
应用实例，访问地址为 http://0.0.0.0:8081
```

浏览器访问 http://127.0.0.1:8081/index.htm，如图所示：

现在你可以向表单输入数据，并提交，如下演示：

文件上传

以下我们创建一个用于上传文件的表单，使用 POST 方法，表单 enctype 属性设置为 multipart/form-data。

index.htm 文件代码修改如下：

```
<html>
<head>
<title>文件上传表单</title>
</head>
<body>
<h3>文件上传：</h3>
选择一个文件上传: <br />
<form action="/file_upload" method="post" enctype="multipart/form-data">
<input type="file" name="image" size="50" />
<br />
<input type="submit" value="上传文件" />
</form>
</body>
</html>
```

server.js 文件代码修改如下:

```
var express = require('express');
var app = express();
var fs = require("fs");

var bodyParser = require('body-parser');
var multer  = require('multer');

app.use(express.static('public'));
app.use(bodyParser.urlencoded({ extended: false }));
app.use(multer({ dest: '/tmp/'}).array('image'));

app.get('/index.htm', function (req, res) {
   res.sendFile( __dirname + "/" + "index.htm" );
})

app.post('/file_upload', function (req, res) {

   console.log(req.files[0]);  // 上传的文件信息

   var des_file = __dirname + "/" + req.files[0].originalname;
   fs.readFile( req.files[0].path, function (err, data) {
        fs.writeFile(des_file, data, function (err) {
         if( err ){
              console.log( err );
         }else{
               response = {
                   message:'File uploaded successfully',
                   filename:req.files[0].originalname
              };
          }
          console.log( response );
          res.end( JSON.stringify( response ) );
       });
   });
})

var server = app.listen(8081, function () {

  var host = server.address().address
  var port = server.address().port

  console.log("应用实例，访问地址为 http://%s:%s", host, port)

})
```

执行以上代码：

```
$ node express_demo.js
应用实例，访问地址为 http://0.0.0.0:8081
```

浏览器访问 http://127.0.0.1:8081/index.htm，如图所示：

现在你可以向表单输入数据，并提交，如下演示：

Cookie 管理

我们可以使用中间件向 Node.js 服务器发送 cookie 信息，以下代码输出了客户端发送的 cookie 信息：

```
// express_cookie.js 文件
var express      = require('express')
var cookieParser = require('cookie-parser')

var app = express()
app.use(cookieParser())

app.get('/', function(req, res) {
  console.log("Cookies: ", req.cookies)
})

app.listen(8081)
```

执行以上代码：

```
$ node express_demo.js
```

现在你可以访问 http://127.0.0.1:8081 并查看终端信息的输出，如下演示：

Node.js RESTful API

什么是 REST？

RREST即表述性状态传递（英文：Representational State Transfer，简称REST）是Roy Fielding博士在2000年他的博士论文中提出来的一种软件架构风格。

表述性状态转移是一组架构约束条件和原则。满足这些约束条件和原则的应用程序或设计就是RESTful。需要注意的是，REST是设计风格而不是标准。REST通常基于使用HTTP，URI，和XML（标准通用标记语言下的一个子集）以及HTML（标准通用标记语言下的一个应用）这些现有的广泛流行的协议和标准。REST 通常使用 JSON 数据格式。

HTTP 方法

以下为 REST 基本架构的四个方法：

- **GET** - 用于获取数据。
- **PUT** - 用于添加数据。
- **DELETE** - 用于删除数据。
- **POST** - 用于更新或添加数据。

RESTful Web Services

Web service是一个平台独立的，低耦合的，自包含的、基于可编程的web的应用程序，可使用开放的XML（标准通用标记语言下的一个子集）标准来描述、发布、发现、协调和配置这些应用程序，用于开发分布式的互操作的应用程序。

基于 REST 架构的 Web Services 即是 RESTful。

由于轻量级以及通过 HTTP 直接传输数据的特性，Web 服务的 RESTful 方法已经成为最常见的替代方法。可以使用各种语言（比如 Java 程序、Perl、Ruby、Python、PHP 和 Javascript[包括 Ajax]）实现客户端。

RESTful Web 服务通常可以通过自动客户端或代表用户的应用程序访问。但是，这种服务的简便性让用户能够与之直接交互，使用它们的 Web 浏览器构建一个 GET URL 并读取返回的内容。

更多介绍，可以查看：RESTful 架构详解

创建 RESTful

首先，创建一个 json 数据资源文件 users.json，内容如下：

```
{
   "user1" : {
      "name" : "mahesh",
      "password" : "password1",
      "profession" : "teacher",
      "id": 1
   },
   "user2" : {
      "name" : "suresh",
      "password" : "password2",
      "profession" : "librarian",
      "id": 2
   },
   "user3" : {
      "name" : "ramesh",
      "password" : "password3",
      "profession" : "clerk",
      "id": 3
   }
}
```

基于以上数据，我们创建以下 RESTful API：

序号	URI	HTTP 方法	发送内容	结果
1	listUsers	GET	空	显示所有用户列表
2	addUser	POST	JSON 字符串	添加新用户
3	deleteUser	DELETE	JSON 字符串	删除用户
4	:id	GET	空	显示用户详细信息

获取用户列表：

以下代码，我们创建了 RESTful API **listUsers**，用于读取用户的信息列表， server.js 文件代码如下所示：

```
var express = require('express');
var app = express();
var fs = require("fs");

app.get('/listUsers', function (req, res) {
   fs.readFile( __dirname + "/" + "users.json", 'utf8', function (err, data) {
       console.log( data );
       res.end( data );
   });
})

var server = app.listen(8081, function () {

  var host = server.address().address
  var port = server.address().port

  console.log("应用实例，访问地址为 http://%s:%s", host, port)

})
```

接下来执行以下命令：

```
$ node server.js
应用实例，访问地址为 http://0.0.0.0:8081
```

在浏览器中访问 http://127.0.0.1:8081/listUsers，结果如下所示：

```
{
   "user1" : {
      "name" : "mahesh",
      "password" : "password1",
      "profession" : "teacher",
      "id": 1
   },
   "user2" : {
      "name" : "suresh",
      "password" : "password2",
      "profession" : "librarian",
      "id": 2
   },
   "user3" : {
      "name" : "ramesh",
      "password" : "password3",
      "profession" : "clerk",
      "id": 3
   }
}
```

添加用户

以下代码，我们创建了 RESTful API **addUser**， 用于添加新的用户数据，server.js 文件代码如下所示：

```
var express = require('express');
var app = express();
var fs = require("fs");

//添加的新用户数据
var user = {
   "user4" : {
      "name" : "mohit",
      "password" : "password4",
      "profession" : "teacher",
      "id": 4
   }
}

app.get('/addUser', function (req, res) {
   // 读取已存在的数据
   fs.readFile( __dirname + "/" + "users.json", 'utf8', function (err, data) {
       data = JSON.parse( data );
       data["user4"] = user["user4"];
       console.log( data );
       res.end( JSON.stringify(data));
   });
})

var server = app.listen(8081, function () {

  var host = server.address().address
  var port = server.address().port
  console.log("应用实例，访问地址为 http://%s:%s", host, port)

})
```

接下来执行以下命令：

```
$ node server.js
应用实例，访问地址为 http://0.0.0.0:8081
```

在浏览器中访问 http://127.0.0.1:8081/addUsers，结果如下所示：

```
{ user1:
   { name: 'mahesh',
     password: 'password1',
     profession: 'teacher',
     id: 1 },
  user2:
   { name: 'suresh',
     password: 'password2',
     profession: 'librarian',
     id: 2 },
  user3:
   { name: 'ramesh',
     password: 'password3',
     profession: 'clerk',
     id: 3 },
  user4:
   { name: 'mohit',
     password: 'password4',
     profession: 'teacher',
     id: 4 }
}
```

显示用户详情

以下代码，我们创建了 RESTful API **:id**（用户**id**）， 用于读取指定用户的详细信息，server.js 文件代码如下所示：

```
var express = require('express');
var app = express();
var fs = require("fs");

app.get('/:id', function (req, res) {
   // 首先我们读取已存在的用户
   fs.readFile( __dirname + "/" + "users.json", 'utf8', function (err, data) {
       data = JSON.parse( data );
       var user = data["user" + req.params.id]
       console.log( user );
       res.end( JSON.stringify(user));
   });
})

var server = app.listen(8081, function () {

  var host = server.address().address
  var port = server.address().port
  console.log("应用实例，访问地址为 http://%s:%s", host, port)

})
```

接下来执行以下命令：

```
$ node server.js
应用实例，访问地址为 http://0.0.0.0:8081
```

在浏览器中访问 http://127.0.0.1:8081/2，结果如下所示：

```
{
   "name":"suresh",
   "password":"password2",
   "profession":"librarian",
   "id":2
}
```

删除用户

以下代码，我们创建了 RESTful API **deleteUser**， 用于删除指定用户的详细信息，以下实例中，用户 id 为 2，server.js 文件代码如下所示：

```
var express = require('express');
var app = express();
var fs = require("fs");

var id = 2;

app.get('/deleteUser', function (req, res) {

   // First read existing users.
   fs.readFile( __dirname + "/" + "users.json", 'utf8', function (err, data) {
       data = JSON.parse( data );
       delete data["user" + 2];

       console.log( data );
       res.end( JSON.stringify(data));
   });
})

var server = app.listen(8081, function () {

  var host = server.address().address
  var port = server.address().port
  console.log("应用实例，访问地址为 http://%s:%s", host, port)

})
```

接下来执行以下命令：

```
$ node server.js
应用实例，访问地址为 http://0.0.0.0:8081
```

在浏览器中访问 http://127.0.0.1:8081/deleteUser，结果如下所示：

```
{ user1:
   { name: 'mahesh',
     password: 'password1',
     profession: 'teacher',
     id: 1 },
  user3:
   { name: 'ramesh',
     password: 'password3',
     profession: 'clerk',
     id: 3 }
}
```

Node.js 多进程

我们都知道 Node.js 是以单线程的模式运行的，但它使用的是事件驱动来处理并发，这样有助于我们在多核 cpu 的系统上创建多个子进程，从而提高性能。

每个子进程总是带有三个流对象：child.stdin, child.stdout 和child.stderr。他们可能会共享父进程的 stdio 流，或者也可以是独立的被导流的流对象。

Node 提供了 child_process 模块来创建子进程，方法有：

- **exec** - child_process.exec 使用子进程执行命令，缓存子进程的输出，并将子进程的输出以回调函数参数的形式返回。
- **spawn** - child_process.spawn 使用指定的命令行参数创建新线程。
- **fork** - child_process.fork 是 spawn()的特殊形式，用于在子进程中运行的模块，如 fork('./son.js') 相当于 spawn('node', ['./son.js']) 。与spawn方法不同的是，fork会在父进程与子进程之间，建立一个通信管道，用于进程之间的通信。

exec() 方法

child_process.exec 使用子进程执行命令，缓存子进程的输出，并将子进程的输出以回调函数参数的形式返回。

语法如下所示：

```
child_process.exec(command[, options], callback)
```

参数

参数说明如下：

command： 字符串， 将要运行的命令，参数使用空格隔开

options ：对象，可以是：

- cwd ，字符串，子进程的当前工作目录
- env，对象 环境变量键值对
- encoding ，字符串，字符编码（默认： 'utf8'）
- shell ，字符串，将要执行命令的 Shell（默认: 在 UNIX 中为 `/bin/sh` ， 在 Windows 中为 `cmd.exe` ， Shell 应当能识别 `-c` 开关在 UNIX 中，或 `/s /c` 在 Windows 中。 在 Windows 中，命令行解析应当能兼容 `cmd.exe` ）

- timeout，数字，超时时间（默认： 0）
- maxBuffer，数字， 在 stdout 或 stderr 中允许存在的最大缓冲（二进制），如果超出那么子进程将会被杀死 （默认: 200*1024）
- killSignal ，字符串，结束信号（默认：'SIGTERM'）
- uid，数字，设置用户进程的 ID
- gid，数字，设置进程组的 ID

callback ：回调函数，包含三个参数error, stdout 和 stderr。

exec() 方法返回最大的缓冲区，并等待进程结束，一次性返回缓冲区的内容。

实例

让我们创建两个 js 文件 support.js 和 master.js。

support.js 文件代码：

```
console.log("进程 " + process.argv[2] + " 执行。" );
```

master.js 文件代码：

```
const fs = require('fs');
const child_process = require('child_process');

for(var i=0; i<3; i++) {
   var workerProcess = child_process.exec('node support.js '+i,
      function (error, stdout, stderr) {
         if (error) {
            console.log(error.stack);
            console.log('Error code: '+error.code);
            console.log('Signal received: '+error.signal);
         }
         console.log('stdout: ' + stdout);
         console.log('stderr: ' + stderr);
      });

      workerProcess.on('exit', function (code) {
      console.log('子进程已退出，退出码 '+code);
   });
}
```

执行以上代码，输出结果为：

```
$ node master.js
子进程已退出，退出码 0
stdout: 进程 1 执行。

stderr:
子进程已退出，退出码 0
stdout: 进程 0 执行。

stderr:
子进程已退出，退出码 0
stdout: 进程 2 执行。

stderr:
```

spawn() 方法

child_process.spawn 使用指定的命令行参数创建新线程，语法格式如下：

```
child_process.spawn(command[, args][, options])
```

参数

参数说明如下：

command： 将要运行的命令

args： Array 字符串参数数组

options Object

- cwd String 子进程的当前工作目录
- env Object 环境变量键值对
- stdio Array|String 子进程的 stdio 配置
- detached Boolean 这个子进程将会变成进程组的领导
- uid Number 设置用户进程的 ID
- gid Number 设置进程组的 ID

spawn() 方法返回流 (stdout & stderr)，在进程返回大量数据时使用。进程一旦开始执行时 spawn() 就开始接收响应。

实例

让我们创建两个 js 文件 support.js 和 master.js。

support.js 文件代码：

```
console.log("进程 " + process.argv[2] + " 执行。" );
```

master.js 文件代码：

```
const fs = require('fs');
const child_process = require('child_process');

for(var i=0; i<3; i++) {
   var workerProcess = child_process.spawn('node', ['support.js', i]);

   workerProcess.stdout.on('data', function (data) {
      console.log('stdout: ' + data);
   });

   workerProcess.stderr.on('data', function (data) {
      console.log('stderr: ' + data);
   });

   workerProcess.on('close', function (code) {
      console.log('子进程已退出，退出码 '+code);
   });
}
```

执行以上代码，输出结果为：

```
$ node master.js stdout: 进程 0 执行。

子进程已退出，退出码 0
stdout: 进程 1 执行。

子进程已退出，退出码 0
stdout: 进程 2 执行。

子进程已退出，退出码 0
```

fork 方法

child_process.fork 是 spawn() 方法的特殊形式，用于创建进程，语法格式如下：

```
child_process.fork(modulePath[, args][, options])
```

参数

参数说明如下：

modulePath： String，将要在子进程中运行的模块

args： Array 字符串参数数组

options：Object

- cwd String 子进程的当前工作目录
- env Object 环境变量键值对
- execPath String 创建子进程的可执行文件
- execArgv Array 子进程的可执行文件的字符串参数数组（默认： process.execArgv）

- silent Boolean 如果为 `true`，子进程的 `stdin`，`stdout` 和 `stderr` 将会被关联至父进程，否则，它们将会从父进程中继承。（默认为：`false`）
- uid Number 设置用户进程的 ID
- gid Number 设置进程组的 ID

返回的对象除了拥有ChildProcess实例的所有方法，还有一个内建的通信信道。

h3>实例

让我们创建两个 js 文件 support.js 和 master.js。

support.js 文件代码：

```
console.log("进程 " + process.argv[2] + " 执行。" );
```

master.js 文件代码：

```
const fs = require('fs');
const child_process = require('child_process');

for(var i=0; i<3; i++) {
   var worker_process = child_process.fork("support.js", [i]);

   worker_process.on('close', function (code) {
      console.log('子进程已退出，退出码 ' + code);
   });
}
```

执行以上代码，输出结果为：

```
$ node master.js
进程 0 执行。
子进程已退出，退出码 0
进程 1 执行。
子进程已退出，退出码 0
进程 2 执行。
子进程已退出，退出码 0
```

Node.js JXcore 打包

Node.js 是一个开放源代码、跨平台的、用于服务器端和网络应用的运行环境。

JXcore 是一个支持多线程的 Node.js 发行版本，基本不需要对你现有的代码做任何改动就可以直接线程安全地以多线程运行。

但我们这篇文章主要是要教大家介绍 JXcore 的打包功能。

JXcore 安装

下载 JXcore 安装包，并解压，在解压的的目录下提供了 jx 二进制文件命令，接下来我们主要使用这个命令。

步骤1、下载

下载 JXcore 安装包 http://jxcore.com/downloads/，你需要根据你自己的系统环境来下载安装包。

1、Window 平台下载：Download，

2、Linux/OSX 下载安装命令，直接下载解压包下的 jx 二进制文件拷贝到 /usr/bin 目录下：

```
$ wget https://s3.amazonaws.com/nodejx/jx_rh64.zip
$ unzip jx_rh64.zip
$ cp jx_rh64/jx /usr/bin
```

将 /usr/bin 添加到 PATH 路径中：

```
$ export PATH=$PATH:/usr/bin
```

以上步骤如果操作正确，使用以下命令，会输出版本号信息：

```
$ jx --version
v0.10.32
```

包代码

例如，我们的 Node.js 项目包含以下几个文件，其中 index.js 是主文件：

```
drwxr-xr-x  2 root root  4096 Nov 13 12:42 images
-rwxr-xr-x  1 root root 30457 Mar  6 12:19 index.htm
-rwxr-xr-x  1 root root 30452 Mar  1 12:54 index.js
drwxr-xr-x 23 root root  4096 Jan 15 03:48 node_modules
drwxr-xr-x  2 root root  4096 Mar 21 06:10 scripts
drwxr-xr-x  2 root root  4096 Feb 15 11:56 style
```

接下来我们使用 **jx** 命令打包以上项目，并指定 index.js 为 Node.js 项目的主文件：

```
$ jx package index.js index
```

以上命令执行成功，会生成以下两个文件：

- **index.jxp** 这是一个中间件文件，包含了需要编译的完整项目信息。
- **index.jx** 这是一个完整包信息的二进制文件，可运行在客户端上。

载入 JX 文件

我们使用 **jx** 命令打包项目：

```
$ node index.js command_line_arguments
```

使用 JXcore 编译后，我们可以使用以下命令来执行生成的 jx 二进制文件：

```
$ jx index.jx command_line_arguments
```

更多 JXcore 功能特性你可以参考官网：http://jxcore.com/。

Wordpress教程

Wordpress 介绍

WordPress是一个开源内容管理系统（CMS），它允许用户构建动态网站和博客。WordPress是基于PHP和MySQL（用于存储数据）。WordPress是最流行的博客系统，在网络上从后端CMS和组件允许更新，定制和管理网站。

什么是内容管理系统（CMS）？

内容管理系统（CMS）是一个软件，它会将所有的数据，如文本，图片，音乐，文档等，在你的网站上管理提供。它可以帮助编辑，发布和修改网站内容。

历史

WordPress 是由马特·查尔斯·穆伦维格和Mike 最初发布于2003年5月27日。WordPress在2009年10月宣布开放源码。

特性

- 用户管理：它允许管理用户信息，例如更换到（订阅者，贡献者，作者，编辑或管理员）的用户角色，创建或删除用户，更改密码和用户信息。用户管理人员的主要部分是身份验证。
- 媒体管理：它用于管理媒体文件和文件夹的工具，您可以轻松地上传，组织和管理网站上的媒体文件。
- 主题系统：它允许修改网站视图和功能。它包括图片，样式表，模板文件和自定义页面。
- 扩展与插件：根据用户需要，有好多插件都可以提供自定义的功能和特性。
- 搜索引擎优化：它提供了多种搜索引擎优化（SEO）工具，这使得 SEO 更简单。
- 多语言： 它允许翻译所有的内容到用户选择的语言。
- 导入器： 它允许导入发布形式的数据。它导入自定义文件，评论，帖子页和标签。

Wordpress优势

- 它是一个开源平台，并免费提供。
- CSS文件可根据设计根据用户需要进行修改。

- 有许多插件和免费提供的模板。用户可以按需要自定义各种插件。
- 它使用所见即所得的编辑器来编辑内容（您看到的就是得到的用户界面，允许用户直接操作的文件布局，而无需布局命令）。
- 媒体文件可以很容易并迅速地进行上载/上传。
- 它提供了多种搜索引擎优化（SEO）工具，这使得SEO简单。
- 根据用户的需要定制是容易的。
- 它允许用户对网站如管理员，作者，编辑和撰稿人创建不同的角色。

Wordpress缺点

- 使用多个插件可以使网站加载和运行沉重。
- PHP知识是必须的，特别在做修改或改变 WordPress 网站时。
- 有时，软件需要更新，以保持 WordPress 的最新在当前浏览器和移动设备可用。WordPress的版本更新会导致数据丢失，因此需要网站的备份副本。
- 修改和格式化图形图像和表格比较困难。

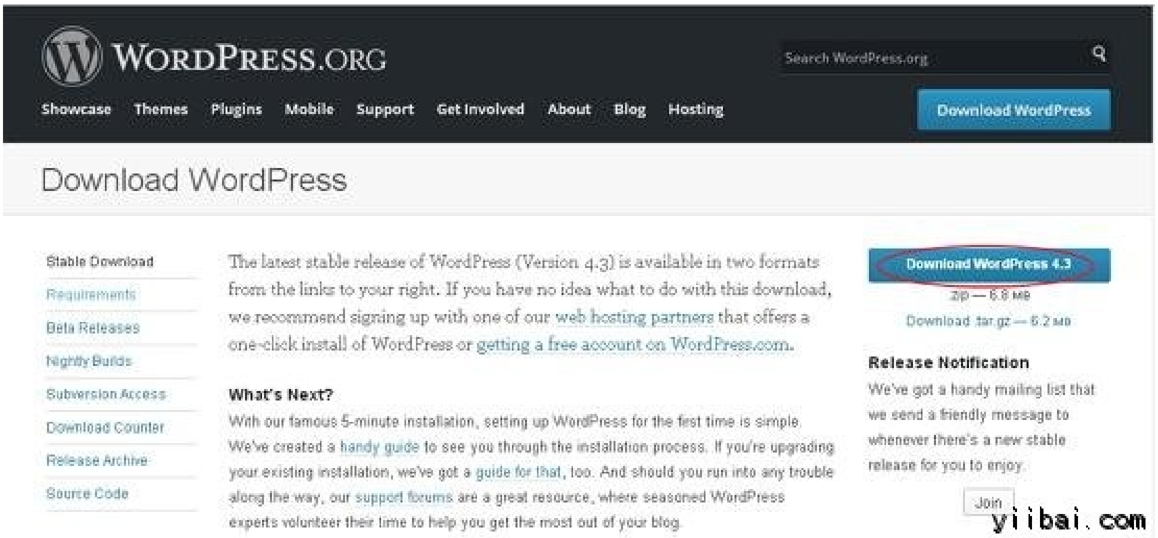
Wordpress安装 - Wordpress教程

WordPress系统要求

- 数据库：MySQL 5.0 +
- Web服务器：
    - WAMP (Windows)
    - LAMP (Linux)
    - XAMP (Multi-platform)
    - MAMP (Macintosh)
- 操作系统: 跨平台
- 浏览器支持：IE (Internet Explorer 8+), Firefox, Google chrome, Safari, Opera
- PHP兼容性：PHP 5.2+

下载WordPress

当打开链接 https://wordpress.org/download/, 将能看到一个屏幕，如下：



请从官方网站WordPress 的 zip文件。

创建存储数据库

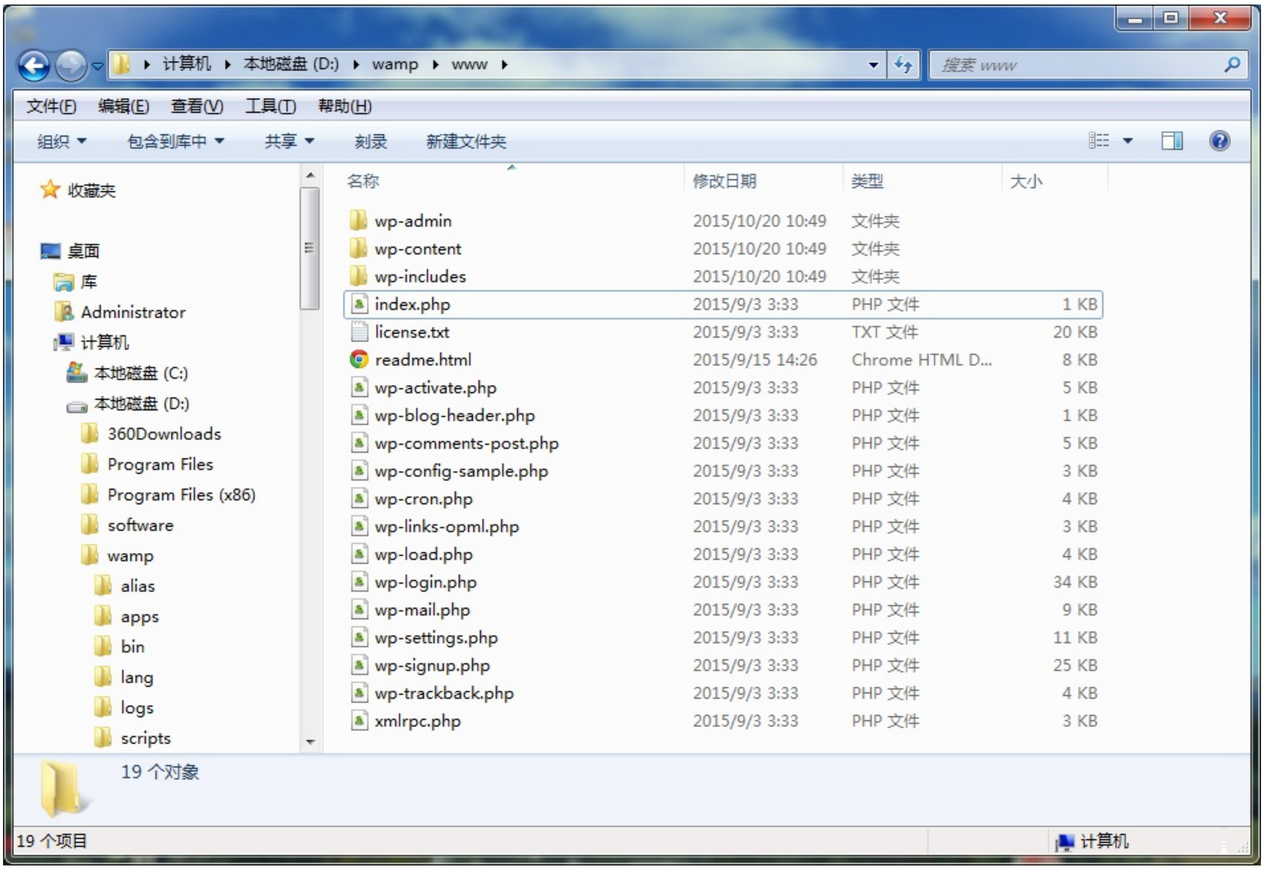
- WordPress需要MySQL数据库。因此，创建用户名/密码（例如用户以"root"，密码为"123456"，也可以设置其它），并创建一个新的空数据库。
- 接下来，您可以继续安装过程如下文所述。

安装向导

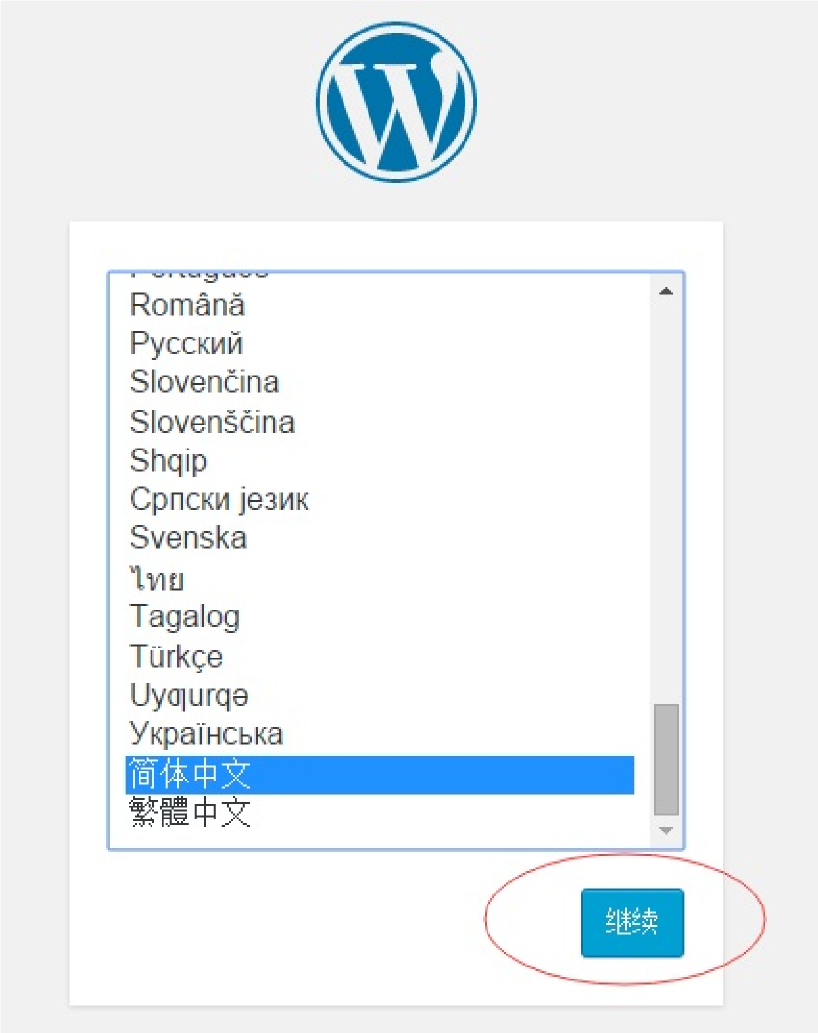
这很容易安装 WordPress 在你的系统上。以下步骤描述了如何设置 WordPress 在本地系统上。

步骤 **(1):** 解压下载的WordPress的文件夹，并把它上传到Web服务器或本地主机，本示例中使用 Wampserver。

步骤 **(2):** 打开浏览器，然后导航到 WordPress 文件路径，那么会得到WordPress的安装程序的第一个屏幕如在下面的屏幕。在我们的示例中，安装的路径是：D:wamp\www 目录，如下：



启动 WAMP，然后在浏览器中输入网址：http://127.0.0.1 ， 如下所示：



为 WordPress 选择语言，然后点击继续。

步骤 (3): 在此步骤中，可以继续使用 WordPress 安装前查看所需的数据库中的信息。



点击 Let's go!

步骤 **(4):** 在这里，有如下的画面描述用来输入关于MYSQL数据库的信息。在操作这一步之前我们先创建一个数据库：yiibai_db。



- 数据库名称：输入您的WordPress要连接的MySQL数据库已经创建的数据库名（这里是：yiibai_db）。
- 用户名：输入您的MySQL数据库的用户名。
- 密码：输入您的MySQL数据库的密码。
- 数据库主机： 填写主机名，默认情况下它会被本地主机（localhost）。
- 表前缀：它被用来添加数据库中表的前缀，这有助于运行多个网站共享同一数据库。它采用默认值。

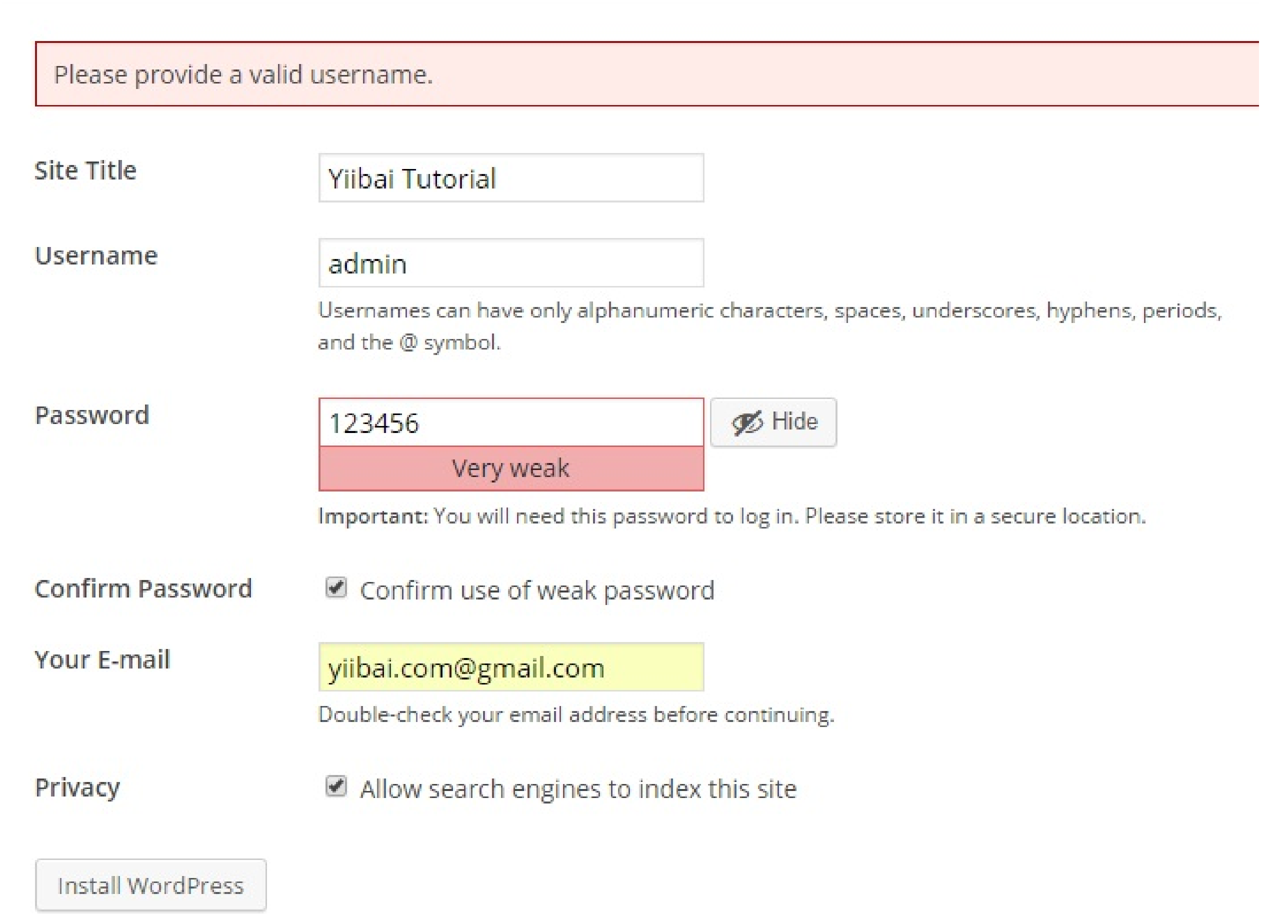
填充的所有信息后，点击 Submit 按钮。

步骤 **(5)**: 在这里，WordPress的检查数据库的设置，让您的确认屏幕，如下图所示。



点周 **Run the install**

Step (6): 在这里，必须输入有关管理的信息。



它包含以下字段：

- **Site Title:** 要创建的 WordPress(网络)的名字。
- **Username:** 输入用户名，当您在登陆WordPress时使用。
- **Confirm Password :** 确认使用弱密码
- **Your E-mail:** 请输入您的电子邮件地址，这有助于恢复密码或任何更新。
- **Privacy:** 它允许搜索引擎索引检查复选框后，这个网站。

填补所有的信息后，点击安装WordPress按钮。

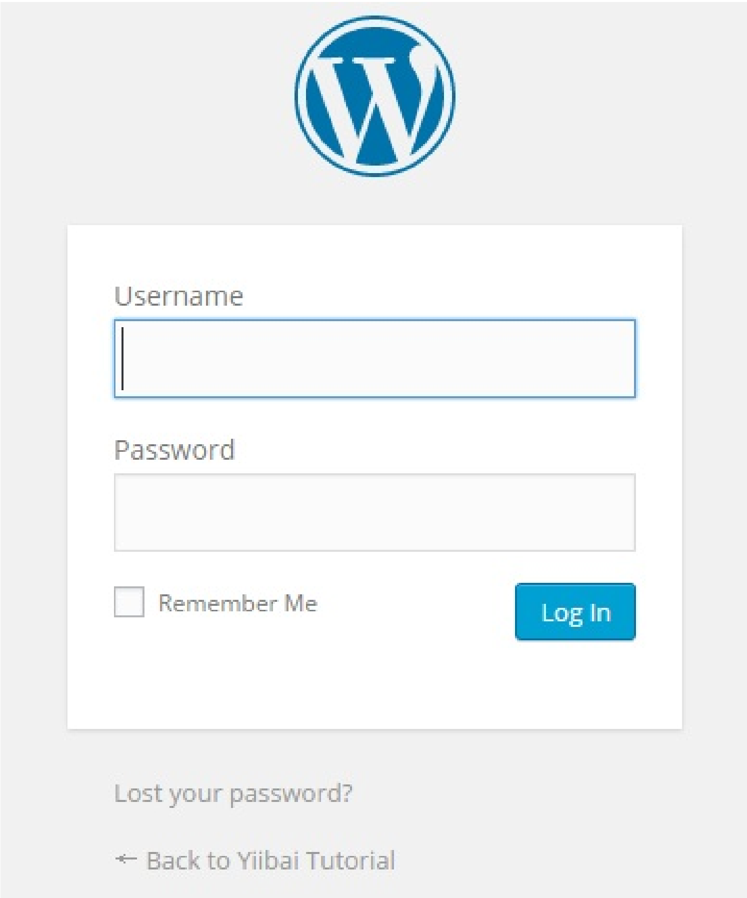
步骤 **(7):** 安装成功后，会得到 WordPress 成功的画面，如下图所示。



在这里你可以查看 WordPress 增加的用户名和密码细节 。

点击 **Log In** 按钮.

Step (8): 点击登录后，会得到一个 WordPress 的管理面板，如下图所示。

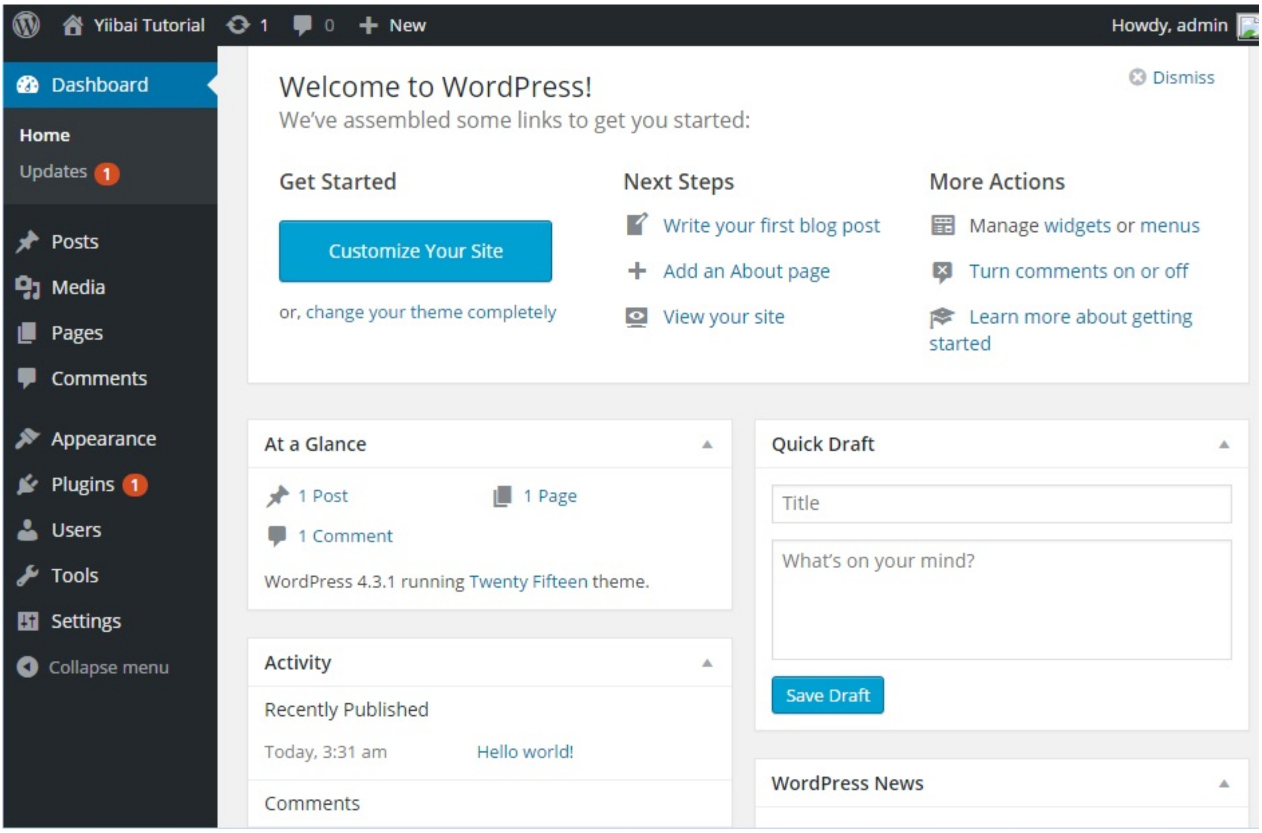


在这里，输入您在安装过程中第6步提到的用户名和密码，点击登录按钮，如图所示。

Wordpress仪表板 - Wordpress教程

WordPress的仪表盘是当登录到你的博客后的管理区域，这将显示该网站的介绍，这是登陆成功后的第一个画面。这些小工具提供信息，并给出了所发生的事情并显示博客在一个一目了然概述的集合。 可以通过使用一些快速链接，如快速写入草案自定义您的需求，回复最新评论等。

仪表盘可分类如下所示。每个类别将在以下部分中讨论：



仪表盘菜单

WordPress 的控制板提供了包含某些菜单选项，如发表帖子，媒体库，页面，评论，外观选项，插件，用户，工具和设置在左侧导航菜单。

Screen Options - 屏幕选项

仪表板包含不同类型的小部件可以显示或隐藏屏幕。它包含复选框，以显示或隐藏屏幕选项，也可以让管理员屏幕上的定制部分选项。

Welcome - 欢迎部分

它包括自定义网站的按钮，允许自定义 WordPress 主题。中央列提供了一些如创建一个博客文章，创建一个页面，查看网站前端的有用的链接。最后一列包含链接到相关的评论窗口小部件，菜单设置，以及在 WordPress 的手抄本链接到 WordPress 的页面等。

Quick Draft -快速草稿

快速草稿是一个小型的帖子编辑器允许书写，保存和发布后的管理仪表板。它包括对草案的标题，草案的一些注意事项，并将其保存为草稿。

WordPress News - WordPress新闻

WordPress新闻插件显示最新消息，如最新的软件版本，更新，警报，新闻就来自WordPress官方博客软件等。

Activity - 活动

活动部件包括博客的最新评论，最近的帖子，最近发布的文章。它可以让取消审核或批准的评论，回复评论，编辑注释也可以删除评论或将其移动到垃圾邮件。

At a Glance - 概览

本节提供了概述博客帖子，发表文章和页面，评论的数量。当点击这些链接，将使用相应的屏幕。它显示目前在网站上运行的主题运行WordPress的最新版本。

Wordpress常规设置 - Wordpress教程

在本章中，我们将研究有关WordPress的常规设置。WordPress的常规设置用于网站设置基本配置。在设置管理屏幕，这是一个默认的设定画面。

要访问常规设置，只要按照下面的步骤：

Step (1): 点击 设置 -> 常规 在WordPress的选项。

Step (2)**: 接着，将显示常规设置**页面如下所示。

以下是设置字段的细节。

- 站点标题**:**它显示在网站中的模板头的名称。
- 副标题**:** 显示你的网站很短的句子。
- **WordPress**地址**(URL):** 这是WordPress的目录的URL，所有核心应用程序文件都存在这里。
- 网站地址**(URL):**输入希望您的网站显示在浏览器的网址。
- **Email** 地址**:**输入您的电子邮件地址，这有助于恢复您的密码或任何更新。
- 成员资格**:**任何人都可以在自己的网站后，选中此复选框注册一个帐号。
- 用户默认角色**:**默认角色被设定为新注册的用户或成员。
- 时区**:**设置基于特定城市的时区。
- 日期格式**:**设置日期格式，需要显示的网站上。
- 时间格式**:**设置时间格式，因为需要显示在网站上。
- 一星期开始于**:**选择您喜欢的位置开始 WordPress 的日历星期的一天。默认情况下设置为周一。
- 站点语言**:**设置WordPress后台显示的语言。

Step(3):填充所有关于常规设置的信息后，点击保存更改按钮。其保存您的所有常规设置信息。

Wordpress撰写设置 - Wordpress教程

撰写设置控制WordPress的写作经验和提供选项来定制WordPress站点。 这些设置控制WordPress 的功能在添加和编辑帖子/发布，网页和文章类型，以及通过电子邮件的可选功能，如远程发布，发布和更新服务。

要访问撰写设置，只要按照下面的步骤：

Step 1: 要转到撰写设置，选择 设置 -> 撰写 选项.



Step 2: 接下来，将显示撰写设置页面，如下图所示。



以下是有关撰写设置字段的细节。

- 默认文章分类目录** : ** 它是被应用到后一个类别，可以把它作为未分类。
- 默认文章形式 : 它由主题选择文章格式应用于后或创造不同风格不同类型的文章。
- 通过电子邮件发布**:**此选项使用电子邮件地址为您创造文章，以及文章发布在您的博客的电子邮件。要使用此功能，需要建立一个秘密的电子邮件帐户并使用POP3访问，并在此地址上接收任何邮件发布的文章。
- 邮件服务器 : 它允许读取发送到Wordpress并存储检索电子邮件。为此，您需要有兼容POP3邮件服务器，这将有URI地址，例如 : mail.example.com，应该在这里输入。
- 登录名 : 要创建文章，WordPress需要自己的电子邮件帐户。登录名将使用这个电子邮件地址，并应当保密的垃圾邮件发送者将帖子链接重定向到自己的网站。
- 密码: 为上述电子邮件地址设置密码。
- 默认邮件分类 : 它允许所有通过电子邮件功能，通过电子邮件发布帖子选择自定义类别。
- 更新服务**:**当发布一个新的文章，WordPress会自动在框中通知网站更新服务。请参阅更新服务关于手抄本可能的服务在一个长长的清单中。

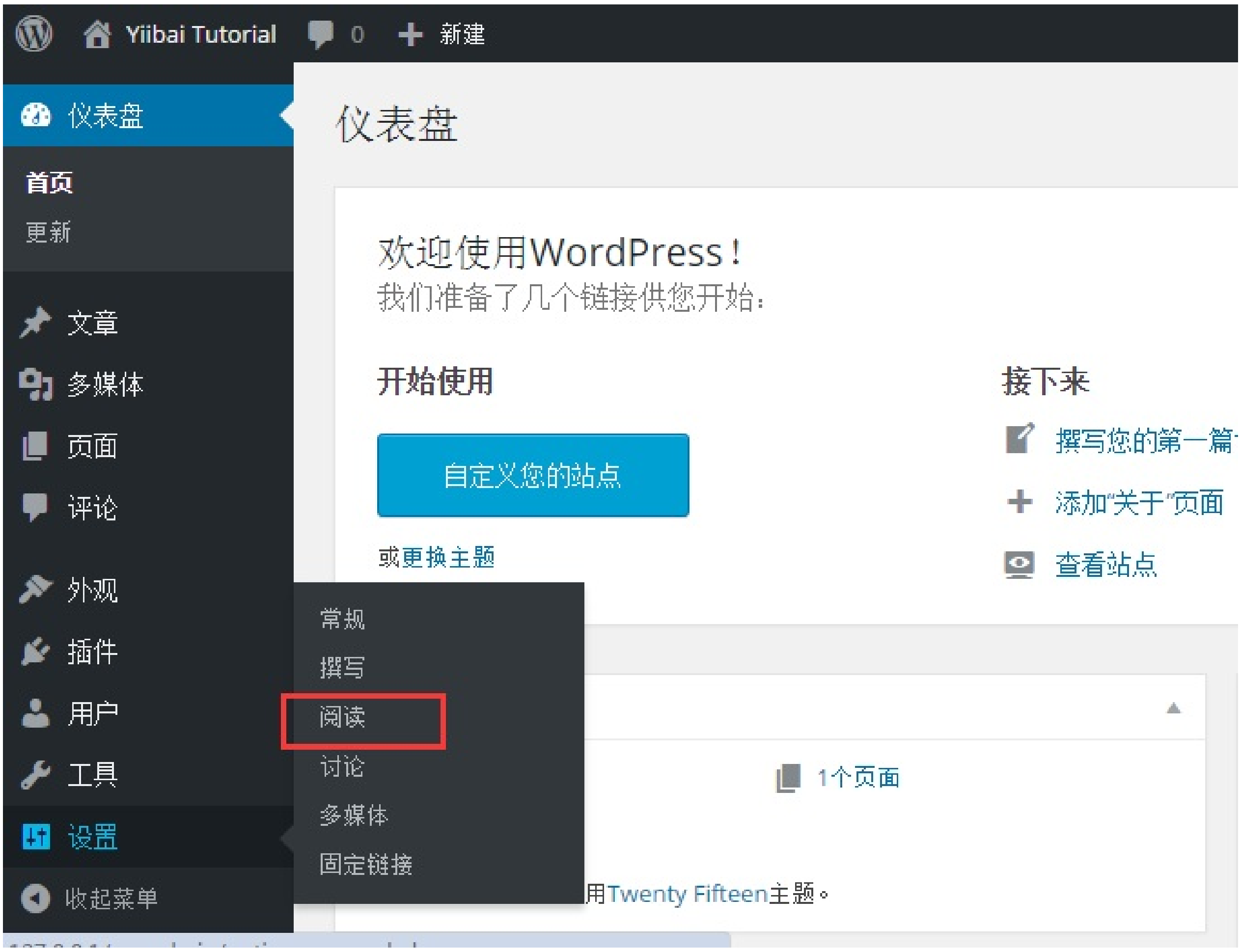
Step(3):填补上述所有资料下一步点击保存更改按钮后，保存您的信息。

Wordpress阅读设置 - Wordpress教程

在本章中，我们将研究有关在WordPress的阅读设置。阅读设置用于设置相关前端页面的内容。在这里你可以设置在主页上显示文章的数量。

要访问的阅读设置，只需遵循以下步骤：

Step (1):点击 设置 -> 阅读 选项在WordPress中.



Step(2):接着，将显示阅读设置页面如下。



以下是阅读设置的字段细节。

- 首页显示**:**这部分是用来显示在以下任何格式的头版：
  - 您的最新文章: 它显示在头版最新的文章。
  - 一个静态页面（下面选择）：它显示在头版上的静态页面。
    - 主页：可以从下拉显示选择想要在头版的实际页面。
    - 文章面: 可以选择其中包含文章的下拉页面。
- 博客页面最多显示： 每个页面或网站上显示文章的数量。缺省情况下，它设置为10。
- **Feed**中显示最近**:**当他们下载网站的feeds，订阅的用户可以查看文章的数目。缺省情况下，它被设置为10。
- 对于**feed**中的每篇文章，显示**:** 此部分用于选择以下任何一种格式来显示日志：
  - 全文: 它显示了完整的文章。这是默认的设置。
  - 摘要: 它显示的文章的摘要。
- 搜索引擎可见性：点击复选框后，不鼓励搜索引擎索引，从这个网站，你的网站会被搜索引擎忽略。

Step(3):填充所有的信息后，接下来点击 保存更改 按钮保存您的阅读设置等信息。

Wordpress讨论设置 - Wordpress教程

在本章中，我们将研究有关在WordPress的讨论设置。WordPress讨论设置可以定义博主与访问者之间的互动。这些设置由管理员做是为了控制通过文章/网页访问的用户。

要访问讨论的设置，只需按照以下步骤操作：

Step 1:点击 设置 -> 讨论选项 在WordPress。

Step 2: 接着，将显示讨论设置页面如下所示。



以下是讨论设置的字段。

- 默认文章设置**:**这些设置是默认创建新页面或新文章。这包含三个设置。 它们是：
    - 尝试通知文章中链接的博客：当您发布的文章则给出通知（发送ping和引用通告）到其他博客。
    - 允许其他博客发送链接通知（pingback和trackback）到新文章: 接受来自其他博客的ping。
    - 允许他人在新文章上发表评论: 可以允许或禁止他人使用该设定你的文章发表评论。

  您可以更改设置，根据您的意愿为个人文章。

- 其他评论设置**:**此设置有以下选项：

  - 评论者必须填写姓名和电子邮件 : 当您选中该复选框，其强制性的访问时填补他们的姓名和电子邮件地址。
  - 用户必须注册并登录才可以发表评论: 如果您选中该复选框，只有已注册那些人才可以发表评论，如果没有选中，任何人都可以发布任何数量的评论。
  - 自动关闭发布x天后的文章上的评论功能: 此选项根据您的意愿可以接受评论仅适用于一个特定的时间段。
  - 启用评论嵌套，最多嵌套x层: 当您选中此选项，游客可以回复或有一个讨论，并得到响应。
  - 分页显示评论，每页显示条评论，默认显示最后最前一页: 如果您的网页有很多评论，那么可以选中此复选框将它们分为不同的页面。
  - 在每个页面顶部显示旧的新的评论: 可以安排升序或降序排列形式的评论。
- 何时发送电子邮件** : ** 此设置包含两个选项，分别是 :
  - 有人发表评论时 ： 当您选中此框中，作者在别人发布每一个评论时会收到电子邮件。
  - 有评论等待审核时 ： 这是用来如果不希望你的评论进行更新之前，它用于放缓管理。
- 在评论显示之前 ： 此设置允许如何您的文章被控制。有两个设置如下 :
  - 评论必须经人工审核 ： 如果您选中此复选框，然后只能由管理员批准的评论可以在文章或页面显示。
  - 评论者先前须有评论通过了审核: 选中时确定批准作者，曾这样评论先前发布的评论的电子邮件地址相匹配的注释进行检查，否则评论等待审核。
- 评论审核 : 包含被允许进入一个注释，链接只是一个具体的数字。
- 评论黑名单 : 可以输入自己的垃圾邮件个词语，不希望访问者进入评论，网址，电子邮件等之后它会过滤评论。
- 头像**:**头像是一个小图像，在你的名字旁边的仪表盘屏幕的顶部右上角显示。这就像你的个人资料图片。在这里有更多的选项，可以设置你的WordPress网站的头像。
  - 头像显示: 除了名字外它显示你的头像。
  - 最高等级: 在这里有替身，可以使用的其他四个选项。它们是G，PG，R和X。这是根据你想显示文章哪种类型选择读者的年龄段。
  - 默认头像: 在这个选项中有几个类型的替身与图像; 可以根据您的访问者的电子邮件地址使用这些替身。

Step 3: 接下来，您可以点击 保存改变 按钮保存更改。

Wordpress媒体设置 - Wordpress教程

在本章中，我们将研究有关在WordPress的媒体设置。它是用来设置高度和宽度，使用到你的网站上的图像。

第**1**步：点击 设置 - > 媒体选择 在WordPress 中。

第**2**步：接下来，你会得到媒体的设置页面，如下图所示。

以下是媒体设置的字段的详细信息

- 缩略图大小: 在这里，您可以设置缩略图的宽度和高度尺寸。
- 中等大小: 在这里可以设置中等大小的图片的宽度和高度。
- 大尺寸: 设置较大的图像的宽度和高度。
- 上传文件: 勾选复选框后，上传的图片会基于文件夹排成年份和月份的形式。

第**3**步：设置在像素尺寸后，点击保存更改按钮。它就可以保存你的媒体设置信息。

Wordpress固定链接设置 - Wordpress教程

在本章中，我们将学习如何在WordPress中设置固定链接。固定链接是一个特定的博客文章或类别使用的永久链接。 它允许设置默认固定链接结构。这些设置用于固定链接添加到您的WordPress文章。下面是访问固定链接设置的步骤。

第**1**步：在左侧导航菜单选项点击 设置 ->固定链接。

第**2**步：当点击固定链接页面出现在屏幕上如下面所示。

这里有一些设置，你可以在WordPress填写设置。

- 常用的设置：

  在这里，可以检查单选按钮，选择您的博客固定链接结构。

  - 默认**:** 在WordPress中它设置默认的URL结构。
  - 日期和名称型**:** 它根据您的文章的日期和名称设置URL结构。
  - 月份和**名称型:**这里月份的名字是URL结构在您的文章中。
  - 数字型**:** URL结构设置数字在您的文章中。
  - 文章名**:** 它设置文章名称在URL结构在您的文章中。
  - 自定义结构**:** 在这里，您可以通过编写所需的名称在给定的文本框中设置您所选择的URL结构。

- 选项**/**可选

  这些是可选的。您可以为主要的类别或标记网址添加自定义结构。如果你的文本框为空，则默认设置将被使用。在这里，你有两个选择。

  - 分类文章：添加自定义分类网址前缀。
  - 标签文章：添加自定义前到您的标签网址。

第**3**步：当用修改完成后，点击保存更改按钮保存设置固定链接。

Wordpress插件设置 - Wordpress教程

在本章中，我们将研究在你的WordPress网站如何使用插件。插件可以很容易地修改，定制或增强的WordPress博客或帖子。WordPress插件是可以上传以扩展网站的功能的软件。它们增加服务或功能到WordPress博客。 插件可让您的工作更轻松。下面是简单添加插件的步骤。

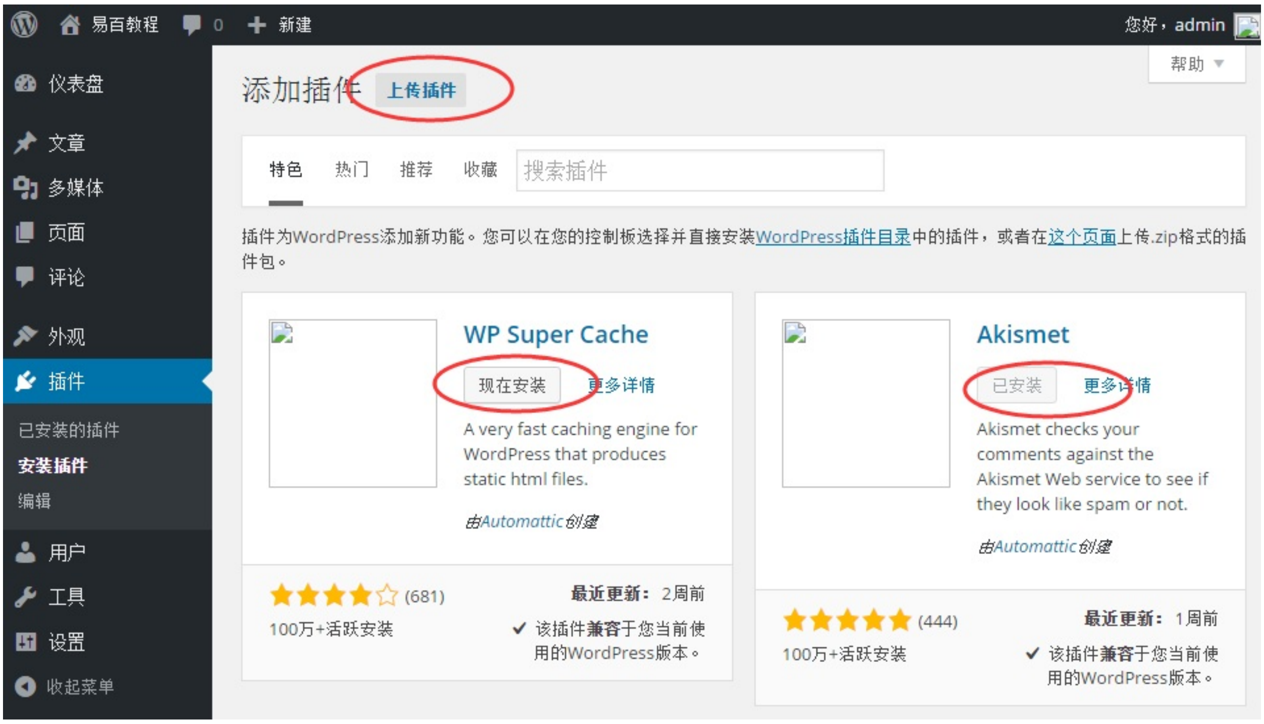
第**1**步：点击左侧侧边栏点击 插件 ->安装插件，如下图所示。



第**2**步：接下来，会出现下面的页面。

在这里，在这个部分中，您可以查看已经安装的插件。

第**3**步：点击 插件->安装插件，如下图所示



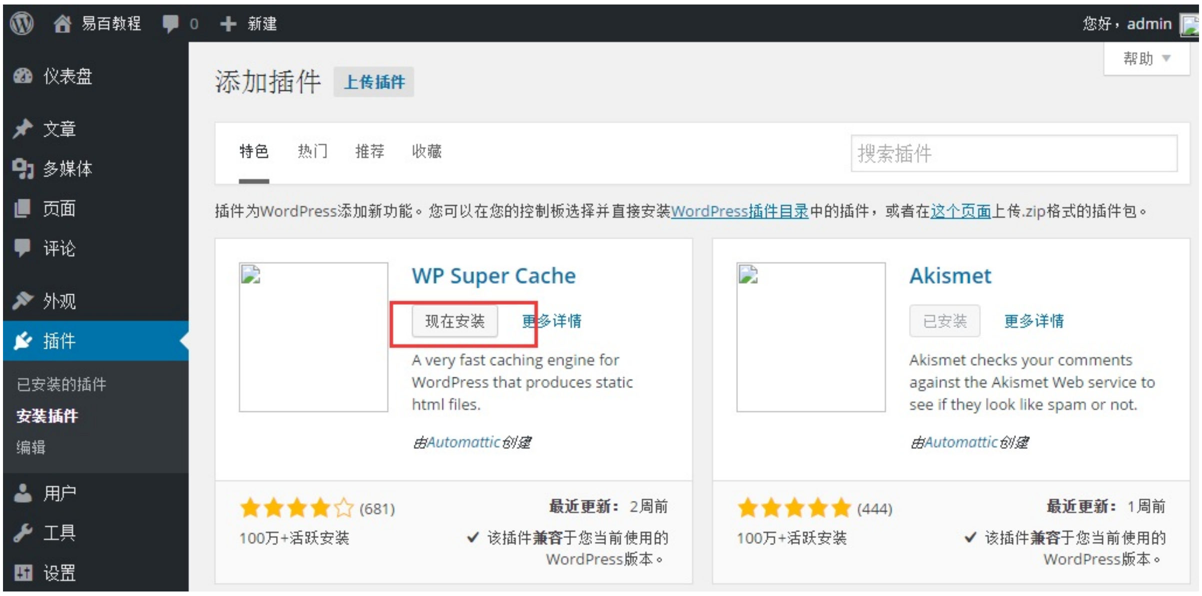
第4步：接下来，WordPress所使用的插件列表就会出现如图。在这里，您可以直接从可用列表中安装插件，也可以点击上传插件。

- 当你点击上传插件，会得到如下页面。



点击浏览需要回到页面，可以从WordPress网站选择插件。如果点击选择文件，可以从你的系统中添加文件。

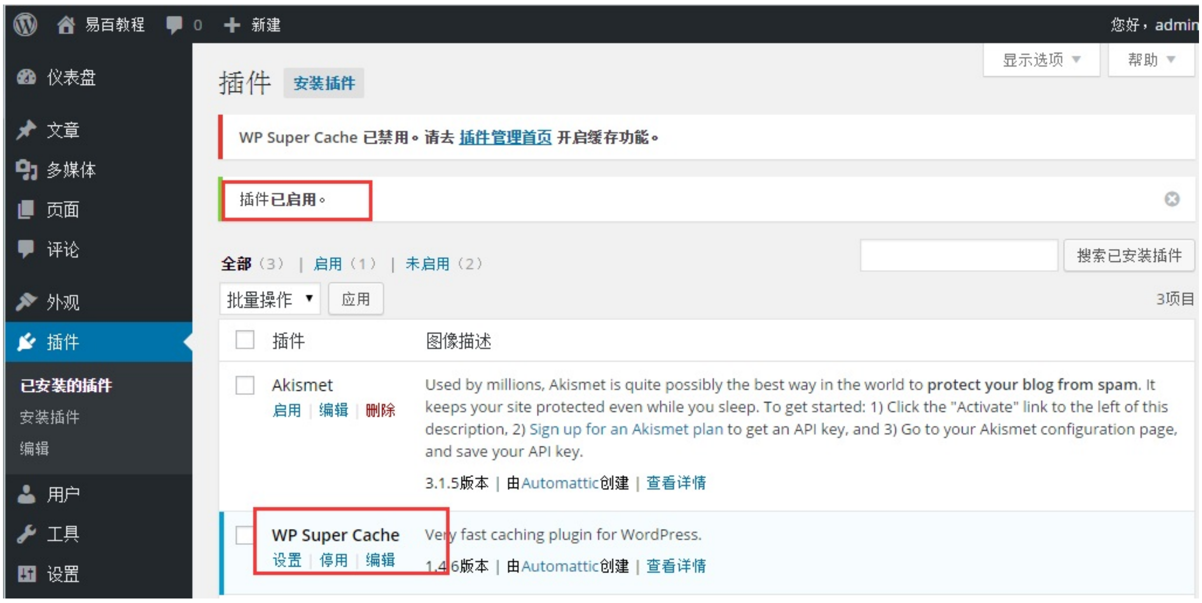
- 否则，在这里我们可以直接选择所需要的插件，然后点击立即安装，如下图所示。



当你点击立即安装，包开始变成下载并安装。后来，点击 启用插件，以活动插件在WordPress中使用，如下图所示。

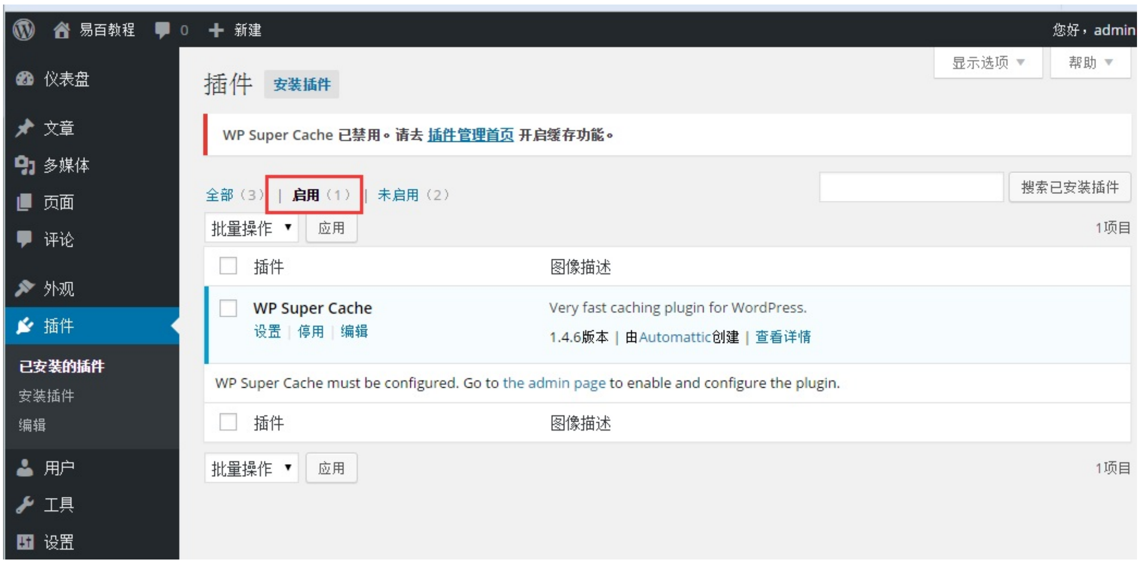


点击启动插件后，会得到一个消息插件激活，也可以在列表中找到已安装的插件。



下面给出的插件激活的消息，可以查看一些选项，比如全部，启用，未启用。

- 当点击启用出现以下页面。在这里，您可以查看所有已启用的插件。



- 当我们点击停用，获取显示其可用但没有被激活的插件。可以通过点击启动激活此插件。



- 点击批量操作，选择选项，然后点击应用按钮，可以更新，删除，启用各个插件。



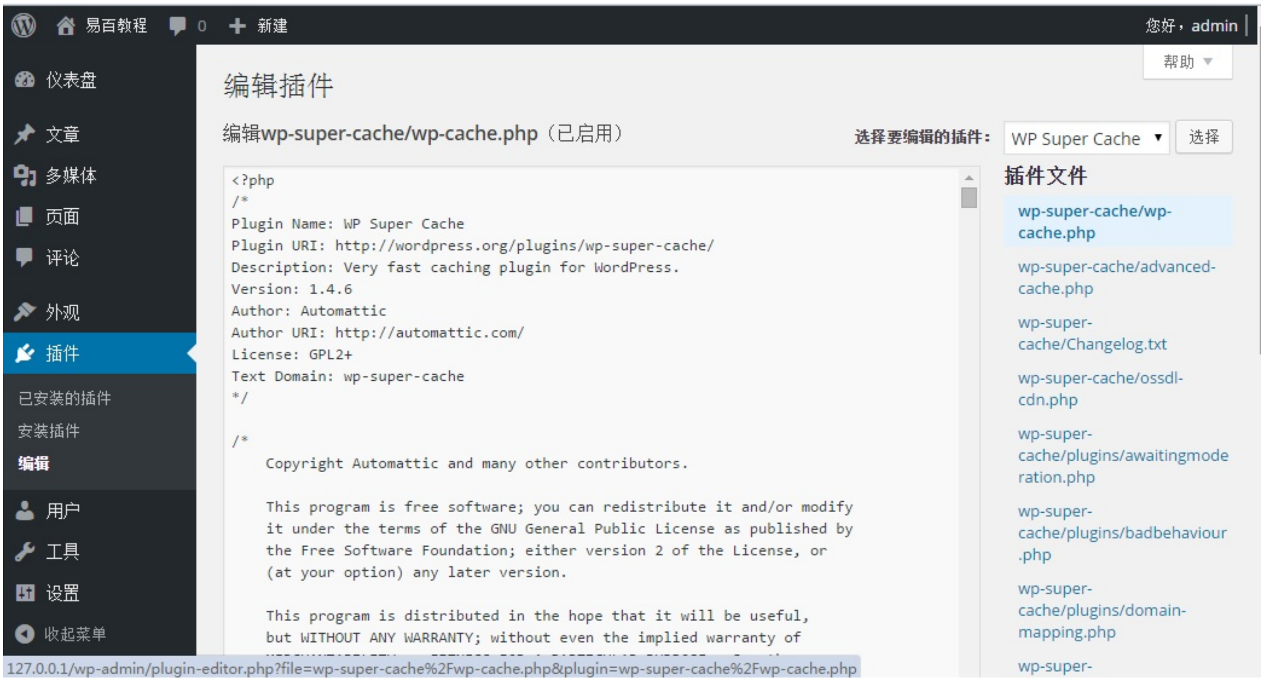
- 在搜索已安装的插件，可以只输入插件名称在文本框，然后点击搜索已安装的插件按钮，当点击搜索安装插件按钮，进入以下页面与相应的插件。这里我们假设搜索：wp 相关插件。



第5步：从边栏点击 插件 ->编辑器。



第6步：接下来，被显示在下面的页面。



此页面允许编辑插件。在这里，我们有一个解释几个选项，如下：

- 选择插件编辑：下面请从下拉列表中选择插件并进行编辑。
- 文档：这里从下拉列表中选择工具来编辑插件。
- 插件文件：在这里，可以从列表中选择文件并编辑。

编辑插件文件完成后，点击更新文件

Wordpress添加分类目录 - Wordpress教程

在本章中，我们将研究如何添加WordPress的分类目录。分类目录用来指示你的网站，将相关文章帖子分类。 它排序分组内容分成不同的部分。这是一个非常方便的方式来组织文章。

要访问分类目录部分，只需遵循以下步骤：

步骤(1)- 在 WordPress中点击 文章 > 分类目录 选项。

步骤（2）：接着，所示显示分类目录页面如下

以下是关于分类目录的字段细节。

- 名称**:** 输入分类目录的唯一名称。
- 别名**:** 它用指定在分类的URL中。
- 父节点**:** 分类目录和标签不同，它可以有层级关系。可以有一个“音乐”分类目录，在这个目录下可以有叫做“流行”和“古典”的子目录。
- 描述**:** 加入分类目录的简要说明。它是任选的。描述只会在一部分主题中显示。

步骤（3）：填写所有关于分类的信息后，单击添加新类别按钮。

步骤（4）： 在点击 添加新分类目录 之后, 如下图所示，新创建的类别将显示在页面右侧中。



Wordpress编辑分类目录 - Wordpress教程

在本章中，我们将学习简单的步骤来编辑WordPress中的分类目录。你可以学习如何在创建分类目录章节在 Wordpress添加分类目录。

以下是简单的步骤，在WordPress中编辑分类目录。

步骤(1)：在WordPress中点击 文章->分类目录。



步骤（2）：接下来，你可以查看 **Category1**（**Category1** 是在 Wordpress添加分类目录 本章中创建）。当光标悬停在分类目录上，然后几个高级选项显示类别名称的下面。进行编辑分类目录有两种方式即 编辑 和 快速编辑：

- 编辑: 单击在类别部分编辑选项，如下图所示。



可以编辑任何必填字段，然后点击更新按钮，如图下面屏幕。



在这里，分类字段与 Wordpress添加分类目录 中的字段相同。

- 快速编辑: 点击快速编辑选项分类目录部分，如下图所示。



在这里，你只能编辑如下图所示的分类目录的名称等，然后点击更新分类按钮。



Wordpress删除分类目录 - Wordpress教程

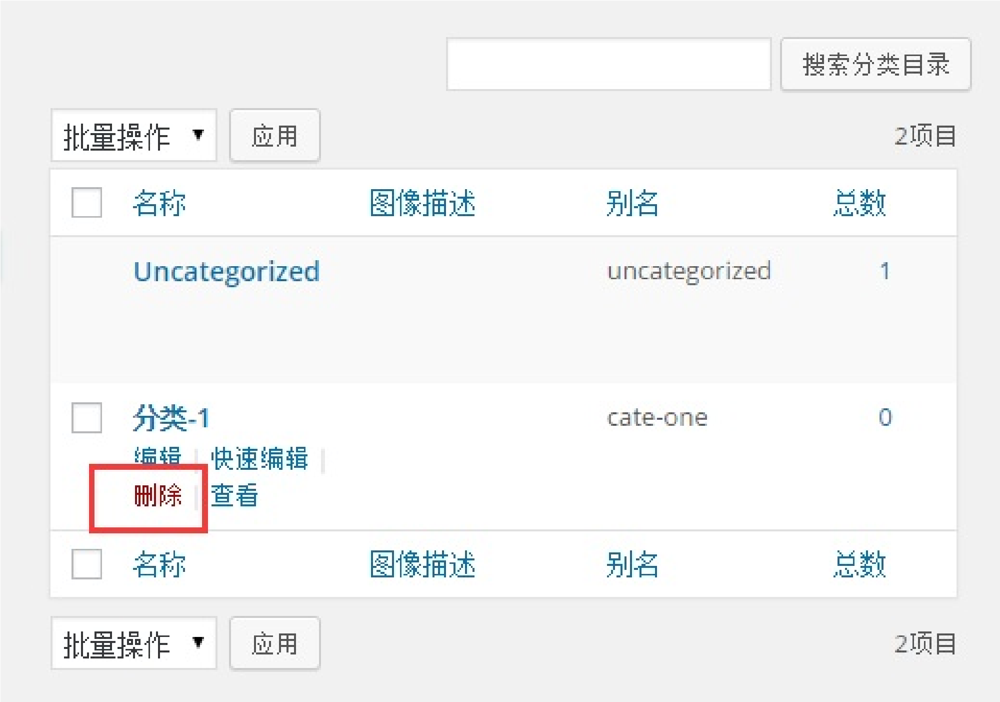
在本章中，我们将学习研究如何在WordPress中删除分类目录。 你可以学习如何创建类别: Wordpress添加目录分类

以下是简单的步骤，用来在 WordPress 删除分类目录。

步骤(1)：在WordPress中点击 文章-> 分类目录。



步骤（2）：接下来，可以删除 分类-1（这是在 WordPress添加分类目录这一章中所创建的）。当光标悬停在类别，然后几个高级选项会显示在类别名称下面。点击删除类别部分，如下图所示。



当点击删除，会得到一个弹出消息，要求确认删除特定分类目录，如下图所示。



您可以点击 确定 按钮，并永久删除分类目录。

WordPress添加/发布文章 - Wordpress教程

在本章中，我们将研究如何在WordPress中添加/发布文章。帖子也被称为文章，有时也被称为博客或博客文章。这些都是用来推广您的博客。

以下是简单的步骤来在WordPress中添加/发布文章。

第1步：在WordPress中点击 文章->写文章。



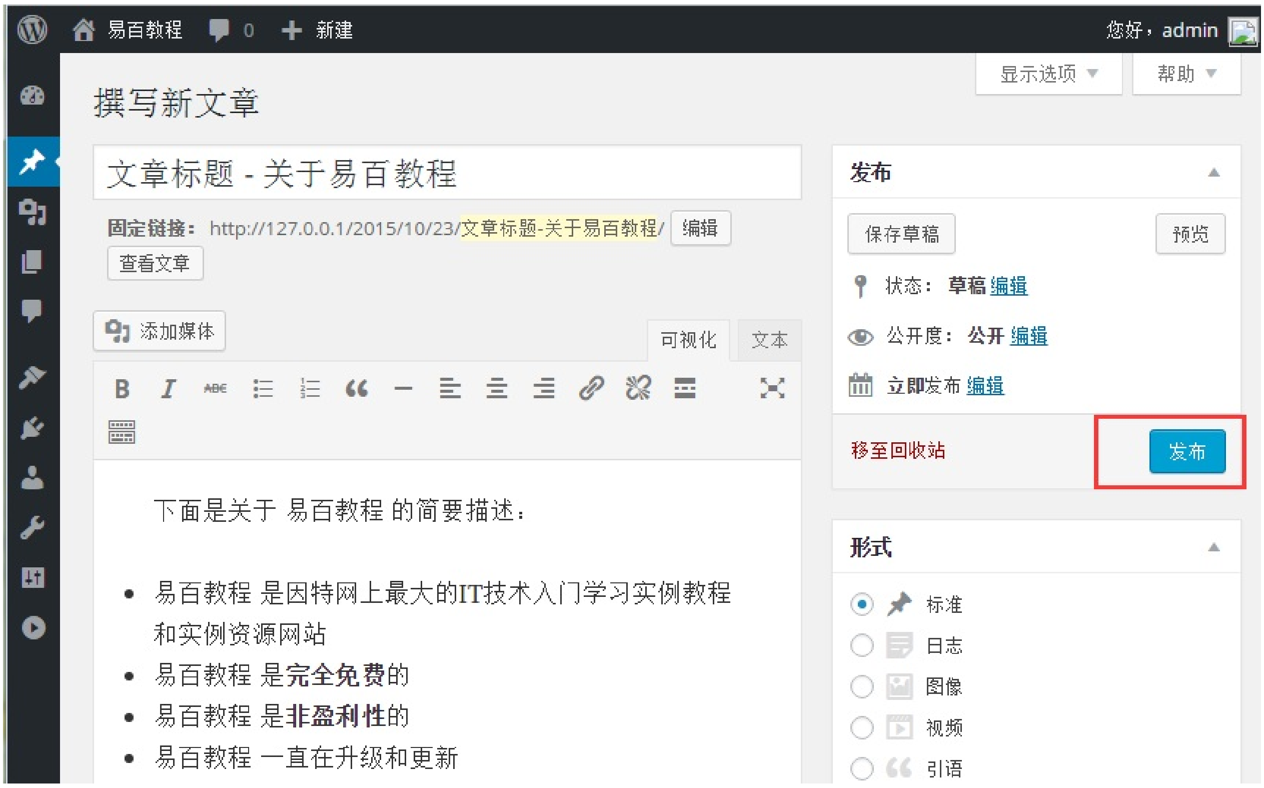
第2步：接下来，将获得文章的编辑页面，如下图所示。可以使用 WordPress 的所见即所得编辑器来添加您的文章内容。我们将学习详细的所见即所得编辑器在 Wordpress添加页面 章节中。



以下是目前在添加文章页面编辑器页面中的字段。

- 文章标题：输入文章标题，如 POST1
- 文章内容：输入文章的内容

第3步：接下来，点击发布按钮来发布的文章。



以下是几个选项会显示在发表部分。

- 保存草稿**:**它保存文章后成稿
- 预览**:**可以在发布前预览您的文章。
- 移到废纸篓**:** 删除文章
- 状态:更改后行列式的状态：发布，待审草案
- 可见度:更改后可见性，即公开，私人或密码保护
- 发表:更改后发表文章的日期和时间

WordPress编辑修改文章 - Wordpress教程

在本章中，我们将学习，如何在 WordPress 中编辑修改文章。可以先学习如何新增文章 Wordpress新添文章

以下是简单的步骤来在WordPress编辑文章/帖子。

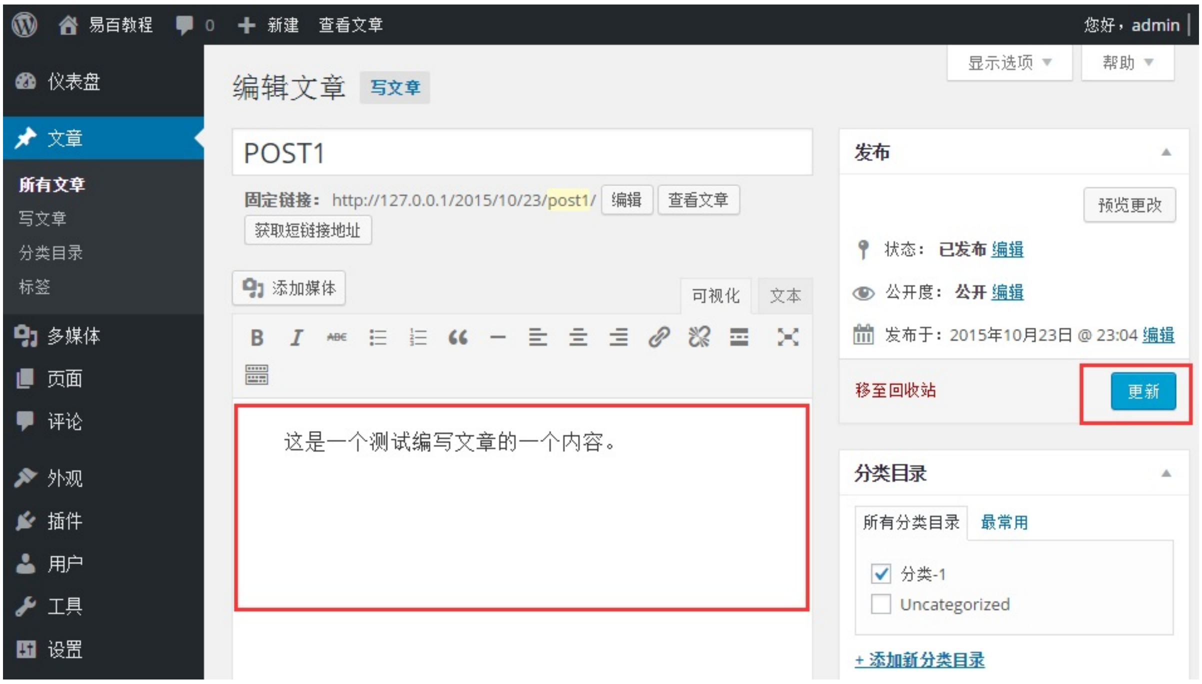
步骤**(1)**：在WordPress中点击 文章 -> 所有文章。

步骤**(2)**：接下来，你可以查看 **Post1** (Post1 是在 Wordpress添加文章 本章创建), 当光标悬停在后，几个高级选项显示在文章名称后面。有两种编辑方式，即：编辑和快速编辑。

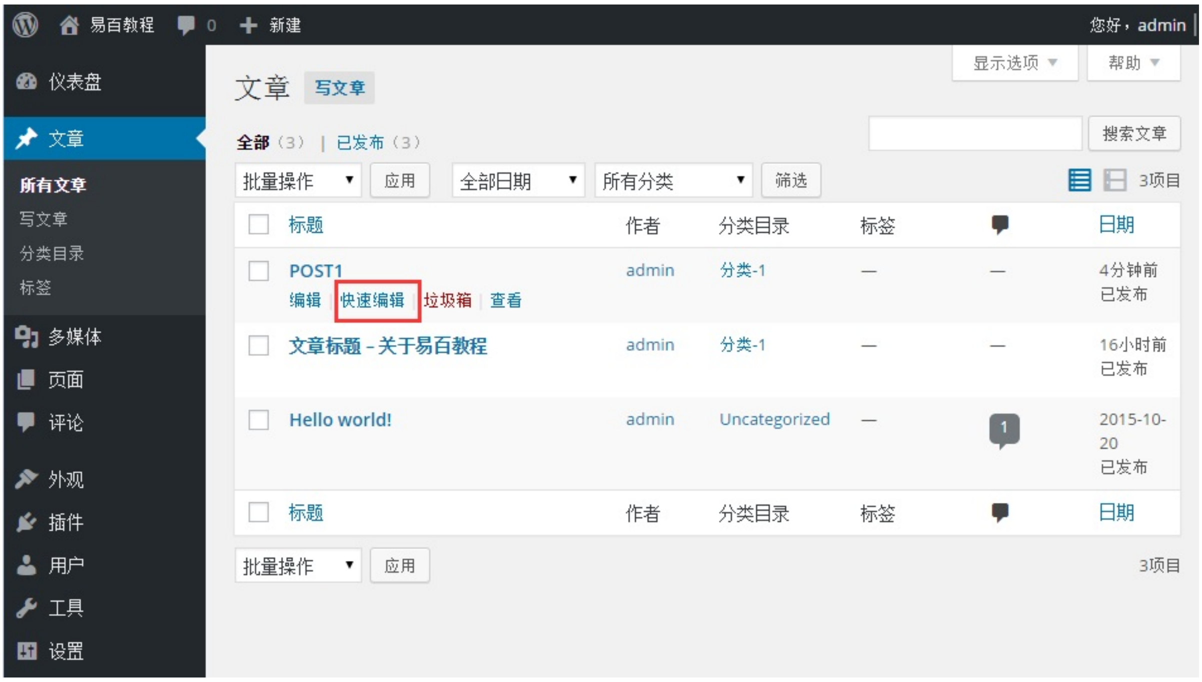
- 编辑：如下图所示，点击 **POST1** 的编辑选项。



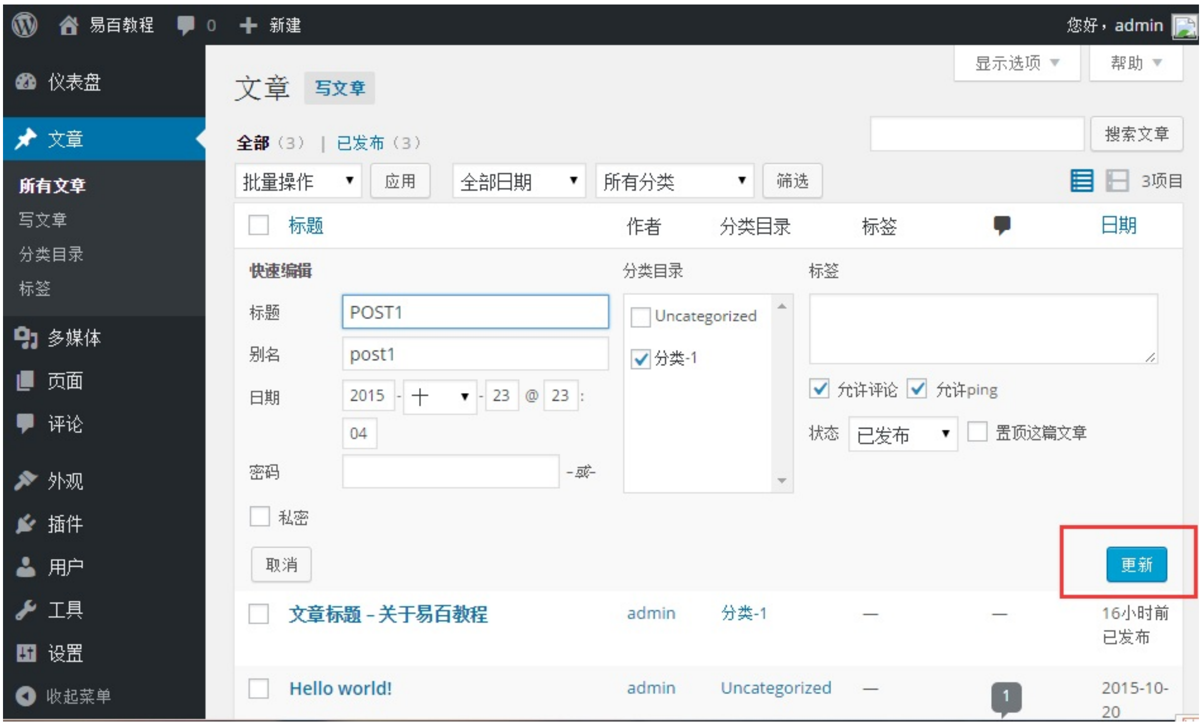
可以按您需要编辑或更改内容或标题之后，然后点击 更新 按钮，如图下面的屏幕。



- 快速编辑：点击在 POST1 的快速编辑选项，如下图所示。



在这里，您可以编辑文章的标题，日期并且还可以选择文章的分类，如下图所示，然后点击更新按钮。



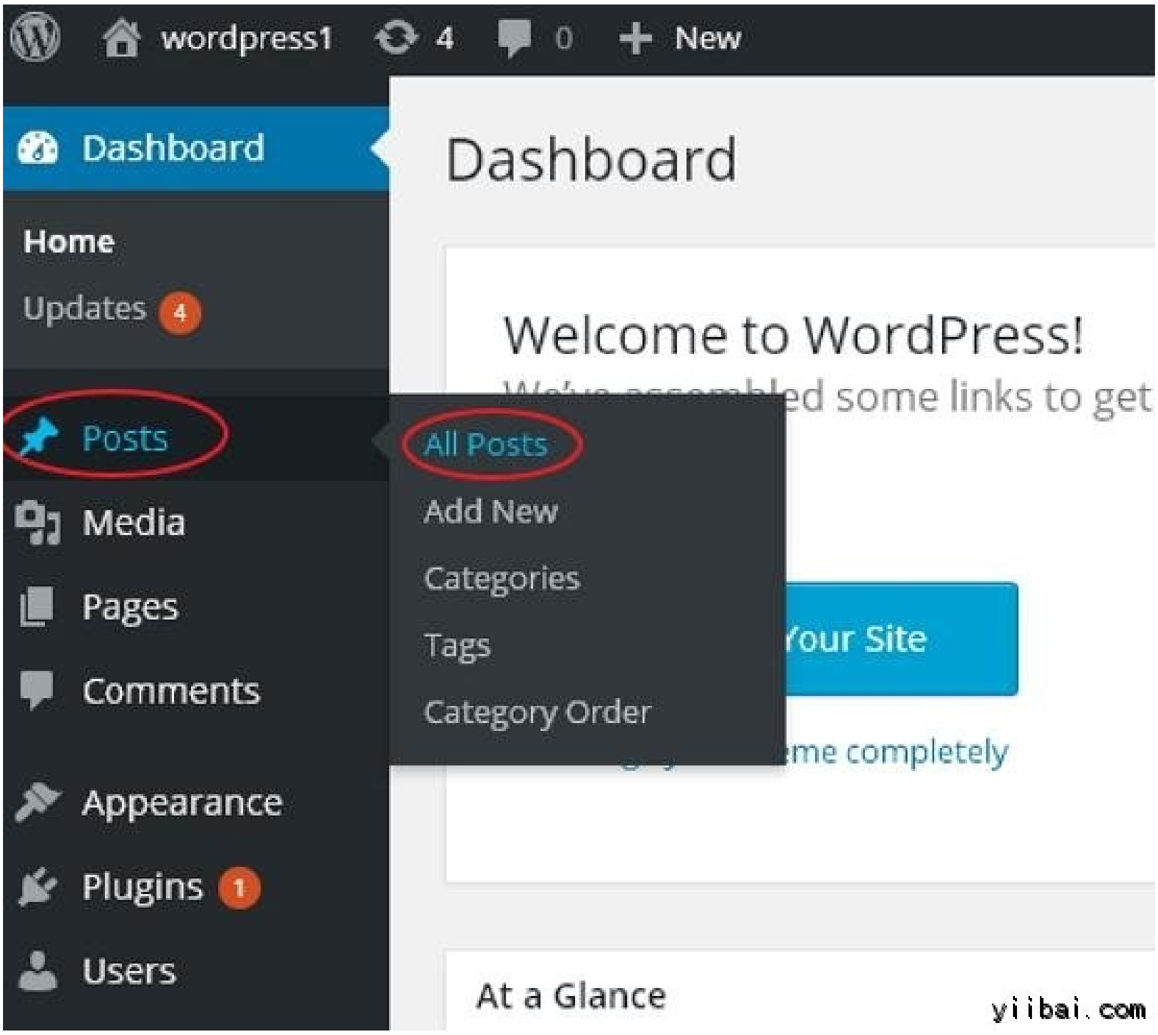
易百教程
新建
您好，admin
仪表盘
文章
所有文章
写文章
分类目录
标签
多媒体
页面
评论
外观
插件
用户
工具
设置
收起菜单
文章 写文章
显示选项
帮助
全部（3）| 已发布（3）
搜索文章
批量操作
应用
全部日期
所有分类
筛选
3项目
标题
作者
分类目录
标签
日期
快速编辑
POST1
别名
post1
Uncategorized
分类-1
允许评论
允许ping
状态
已发布
置顶这篇文章
密码
私密
取消
更新
文章标题 - 关于易百教程
admin
16小时前
已发布
Hello world!
2015-10-20

Wordpress删除文章 - Wordpress教程

在本章中，我们将学习如何在WordPress中删除文章。可以学习如何添加文章在 Wordpress 添加文章 章节。

按照以下步骤在WordPress删除文章。

步骤(1)：在Wordpress中点击 文章 -> 所有文章。



步骤（2）：接下来可以删除POST1（POST1是在 WordPress添加文章 这一节中创建的），当光标悬停在后，在 POST1 下方显示那几个高级选项。点击 垃圾箱 选项删除该文章。



步骤（3）：接下来，可以查看您的文章列表以确认上面的帖子被删除。



Wordpress预览文章 - Wordpress教程

在本章中，我们将学习如何在WordPress中预览文章。预览文章是查看这篇文章发布给用户之前。安全预览文章，并确认在看后是否要发布到网站上。可以根据您的需要编辑在预览后更改后。

以下是简单的步骤来在WordPress中预览文章。

步骤**(1)**：在WordPress中点击 文章 -> 所有文章。

或者可以直接查看您的文章，当编辑时点击预览按钮添加后如下图所示



步骤（3）：接下来，当你点击观看或预览，可以查看您的文章如下图所示。



Wordpress发布文章 - Wordpress教程

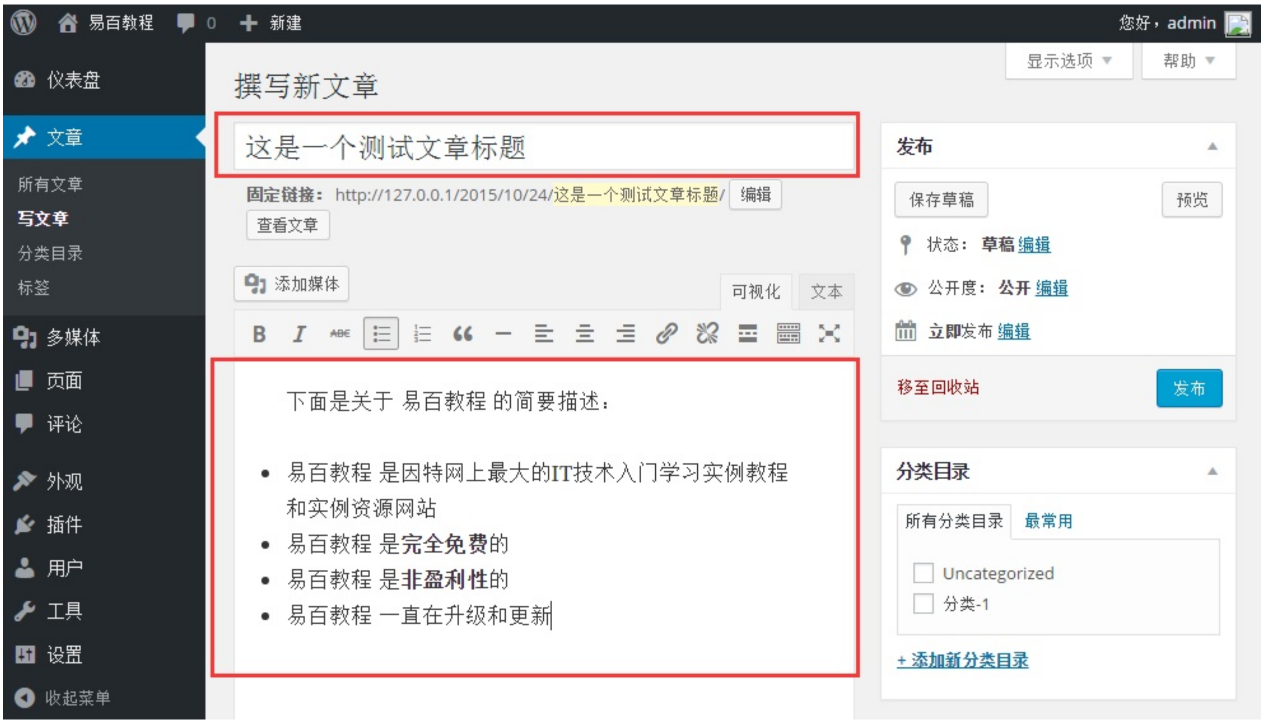
在本章中，我们将学习如何在WordPress中发布文章。发布是用来提供给所有用户，每个用户都可以查看某个文章的发布。在WordPress发布一个新的文章是一个简单的过程。

下面是在WordPress发布文章的步骤。

第**1**步：在 WordPress 中点击 文章 -> 写文章。



第**2**步：接下来，将获得文章的编辑页面，如下图所示。您可以使用 WordPress 的所见即所得的编辑器添加/编写文章的内容。



第3步：接下来，点击 发布 按钮来发布文章。



点击 发表 后您的文章得以发表，供用户查看。

WordPress媒体库 - Wordpress教程

在本章中，我们将了解 WordPress 中的媒体库。媒体库包括图像，音频，视频和文件，可以写一个文章或网页时上传并添加到内容中。在这里可以看到，添加，编辑或删除任何媒体相关的对象如果没有必要时。

下面是了解有关媒体库的步骤。

步骤**(1)**：在 WordPress 中点击 多媒体-> 媒体库。



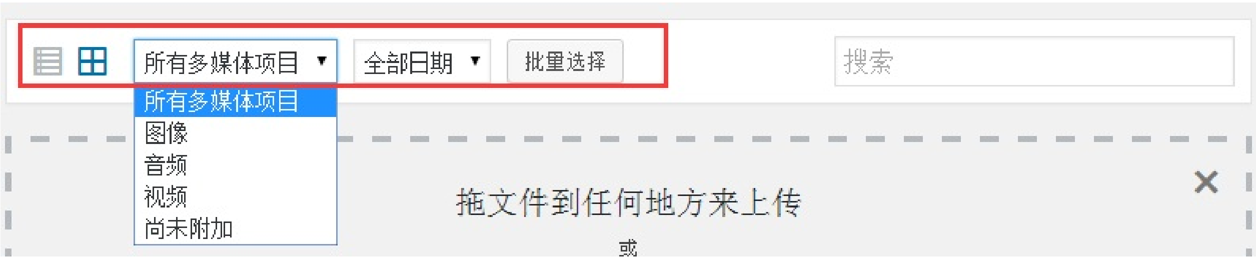
步骤（**2**）：接下来，可以看到如图像，音频，视频媒体文件。单击 添加 媒体按钮。



步骤（3）：接下来，获取显示上传新媒体页面。可以了解如何添加媒体在 WordPress添加媒体 。



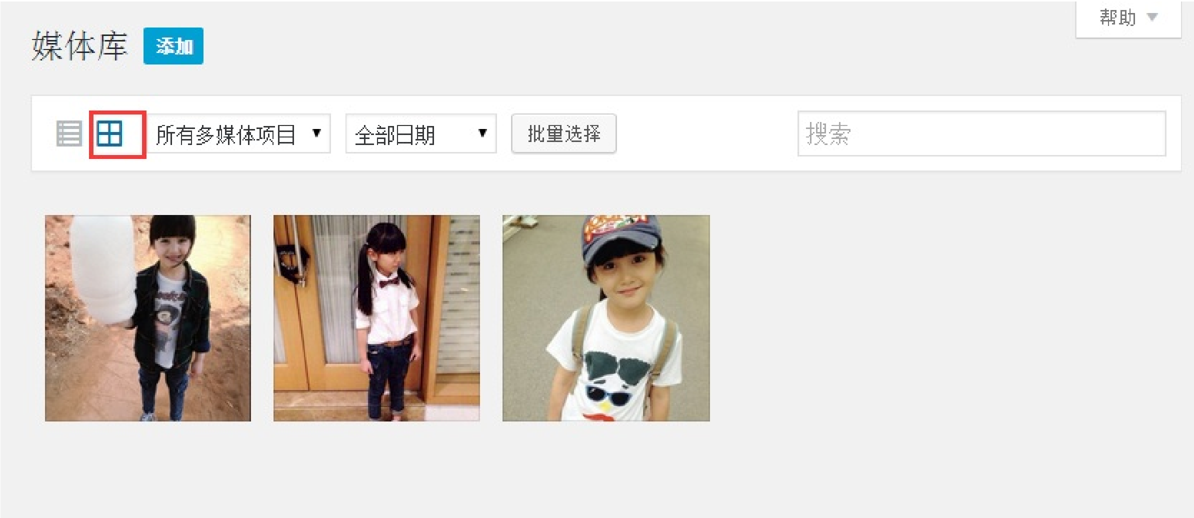
步骤（4）：接下来，可以看到一些工具条，如下图所示。



- 列表视图：显示图像和视频在列表形式。



- 网格视图：以网格格式显示所有图像，如下图所示。



- 过滤图像和视频它过滤图像和视频。
- 搜索框：通过将名称在框中搜索特定的图像。

Wordpress添加媒体 - Wordpress教程

在本章中，我们将学习如何在 WordPress 中添加媒体文件。WordPress允许你添加类似视频，音频和图像的媒体文件。

按照以下步骤来添加媒体。

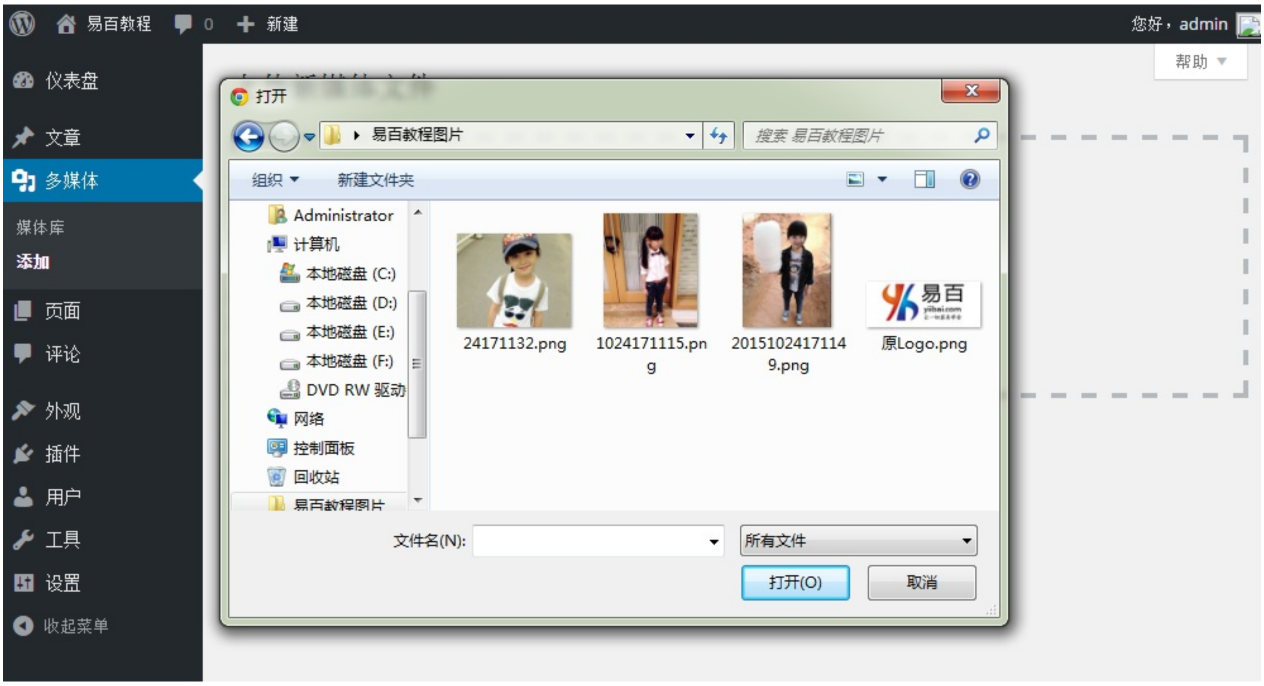
步骤(1)：在WordPress中点击 媒体 -> 添加。



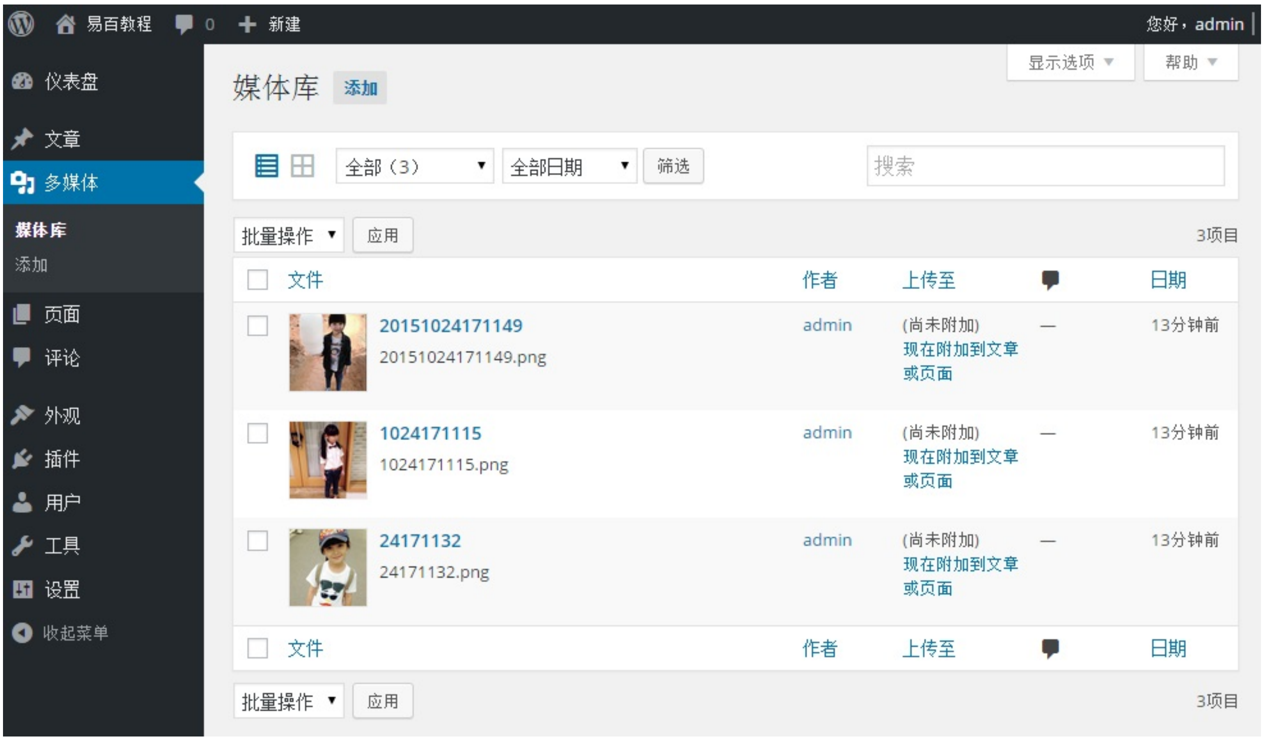
步骤（2）：接下来，点击 选择文件 选项从本地存储选择，如下图所示的文件。



步骤（3）：接下来，选择 添加媒体文件 如下图所示，如图像和音频。



步骤（4）：接着，可以查看添加后的媒体文件的列表，如下所示。



WordPress插入媒体 - Wordpress教程

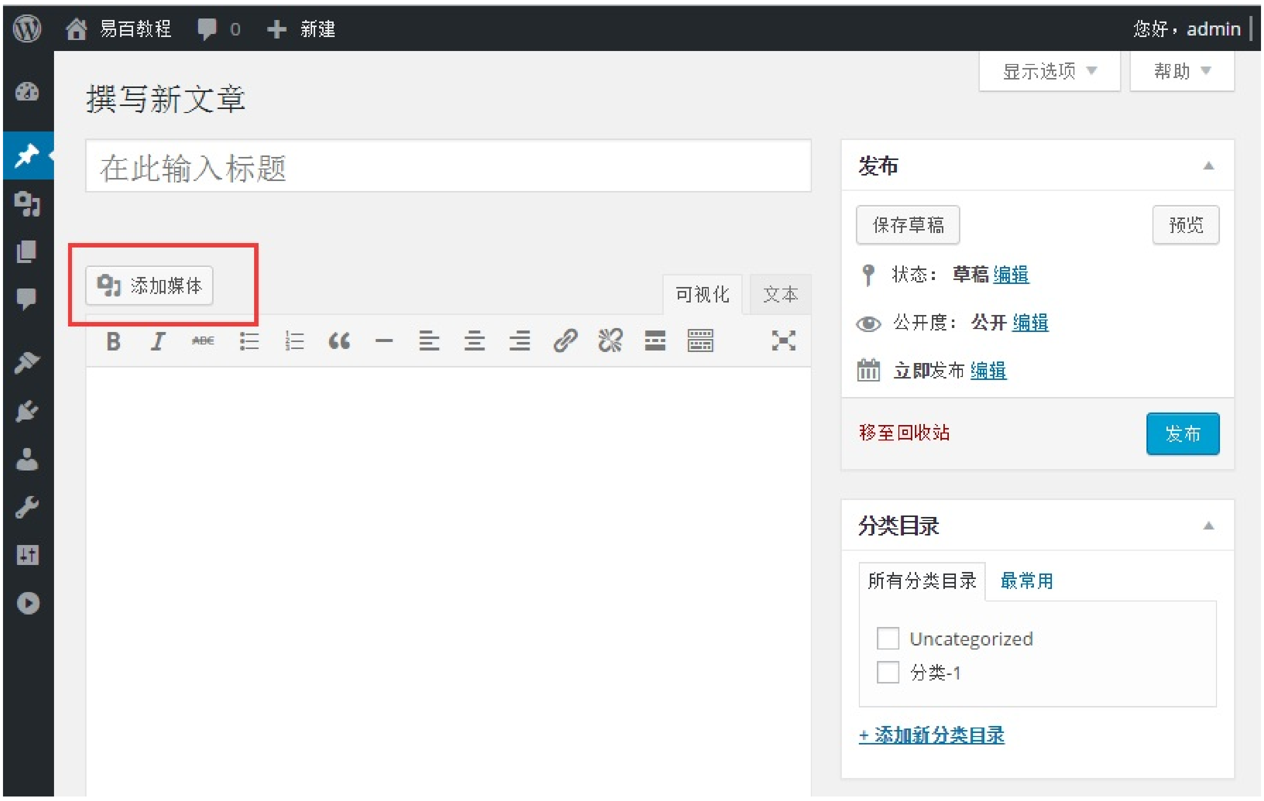
在本章中，我们将学习如何在WordPress中插入传媒。媒体文件可以从本地存储或从网址插入到您网页或文章。

按照以下在 WordPress 中插入媒体的步骤。

步骤**(1)**：在 WordPress 点击 文章 -> 写文章



步骤（**2**）：接下来，单击 添加媒体。



步骤（3）：可以从媒体库选项卡中选择文件，如下图所示。



关于所选择的媒体文件的信息，将显示在屏幕右侧的附件详细下。单击 插入至文章，图像将被插入到文章。在附件详细信息部分，可看到如URL，标题，图片说明，替换文本和描述的图像信息。或者可以直接从系统通过对已上传文件选项卡插入图片。点击 插入至文章 按钮。


插入多媒体
创建相册
特色图片
从URL插入
插入多媒体
上传文件
媒体库
拖文件到任何地方来上传
或
选择文件
最大上传文件大小：2 MB。
已选1个
清空
插入至文章

WordPress编辑媒体 - Wordpress教程

在本章中，我们将学习如何编辑WordPress中的媒体。可以管理所有关于你的媒体，保存在媒体库中的信息。

以下是WordPress中编辑媒体的步骤。

步骤(1)：点击 多媒**体 -> 媒体**库，然后单击媒体项目或编辑链接的名称。



步骤（2）：浏览的媒体文件列表。选择任意一个图像编辑。



步骤（3）：接下来，可以在右侧几个选项查看编辑媒体页面。



- URL**:** 可以从一个链接读取媒体文件。
- 标题**:** 这显示了媒体的名称。标题通常以画廊和附件页面上显示，如果主题或插件设计显示它。
- 替代文本**:** 图像的 alt 文本，这是用于描述媒体。用于可用性。
- 图像描述：媒体文件的解释。

- 永久删除：永久删除您的媒体文件。

步骤（4）：完成编辑所需的字段后，图像所做的更改会自动保存。



WordPress添加页面 - Wordpress教程

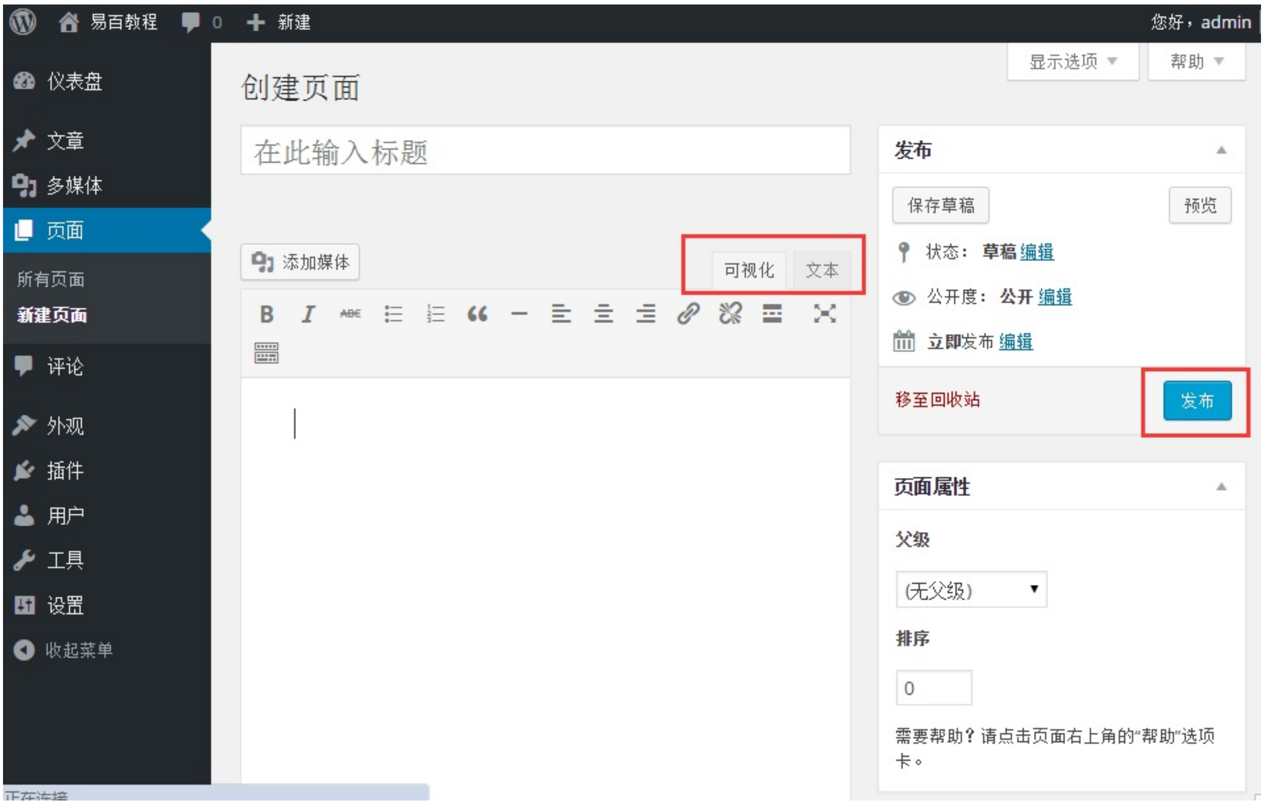
在本章中，我们将学习如何在WordPress中添加页面。添加页面类似于WordPress的增加文章。页面是静态内容往往不是经常改变其显示的信息。

以下是在WordPress中添加页面的简单步骤。

步骤(1)：点击 页面 -> 新建页面，如下图所示。



步骤（2）：接下来，将得到编辑的页面，如下图所示。在下面的图片可以看到，在编辑器页面有两个标签，视觉和文本。在这里可以插入两种文本。在这里，我们将学习有关将文本转换成可视格式。



以下是在添加新页面编辑器页面中的字段的详细信息。

标题

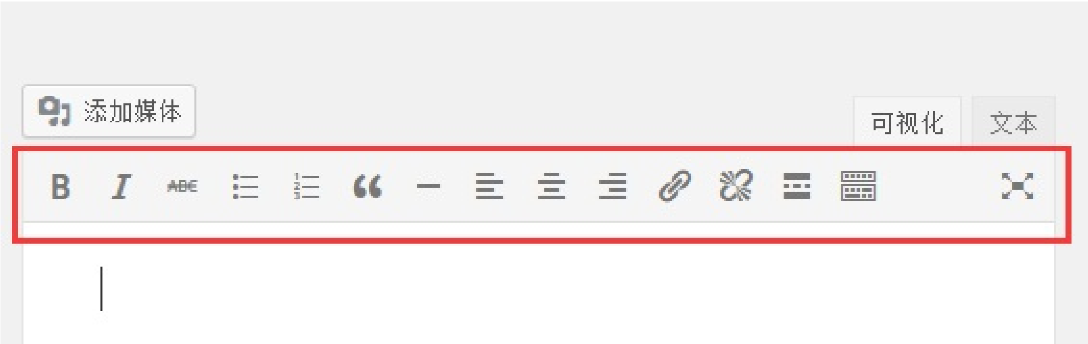


它用于写文章，这是在页面上之后显示标题。

固定链接示出了用于下文的标题的页面潜在的网址。这个 URL 是按照给定的标题生成。

WYSIWYG 编辑器

这是一个所见即所得的编辑器，这类似我们编辑文章的内容文字处理器接口。



下面是目前所见即所得的编辑器的选项

- 粗体按钮：用于加粗字体。
- 斜体：用于斜体字体。
- 项目符号列表：它增加了项目符号的内容。
- 编号列表：这可以添加数字内容的列表。
- 大段引用：它引用文字。
- 水平分割线：创建一个横线在句子之间。
- 左对齐：将在页面左侧对齐内容。
- 右对齐：设置内容的页面右侧对齐。
- 对齐：设置在页面的中心对齐内容。
- 添加链接：它允许添加一个链接到您的内容。当点击这个按钮，显示以下页面。



文本插入

它用于写文章内容。

发布

发布页面到网站上给用户。

页面属性

页面属性模块允许您选择的父级页面为特定页面。还可以设置顺序页面。

- 父级：允许您选择父页面。



- 顺序：设置页面的顺序。

特色图像

它包括在网页中的图像。

WordPress发布页面 - Wordpress教程

在本章中，我们将学习如何在WordPress发布页面。 发布提供给所有用户，使得每个用户可以查看该特定页面的页面。在WordPress发布一个新的页面是一个简单的过程。

按照以下步骤来在WordPress中发布页面。

第**1**步 ：在 WordPress 中点击 页面 -> 新建页面。

第**2**步 ：接下来得到编辑器，如下图所示。可以使用 WordPress 的所见即所得编辑器来添加您的网页的实际内容。



第**3**步 ：如下图所示接下来，点击发布按钮。



点击后 发布 后您的文章得以发表，供用户查看。

WordPress编辑页面 - Wordpress教程

在本章中，我们将研究如何来编辑 WordPress 的页面。你可以学习如何添加页面在 WordPress添加页面。

步骤(1)：在 WordPress 点击 页面 ->所有页面，如下图所示。

步骤（2）：接下来，可以查看关于我们（关于我们已在 WordPress添加页面 这一章中创建）。当光标悬停在页面，然后选择几个选项得到下面 关于我们 的显示。之后编辑它们，即编辑和快速编辑两种方式。

- 编辑：点击编辑选项关于我们如下图所示。



可以编辑或更改网页内容或标题按，然后点击更新按钮，如下面的图。



- 快速编辑：单击快速编辑选项关于我们如下图所示。

在这里，您可以编辑标题和关于我们页面的日期，也可以选择父页面，如下图所示，然后点击更新按钮。



WordPress删除页面 - Wordpress教程

在本章中，我们将学习如何在WordPress中删除页面。你可以学习如何添加页面在WordPress添加页面章节 。

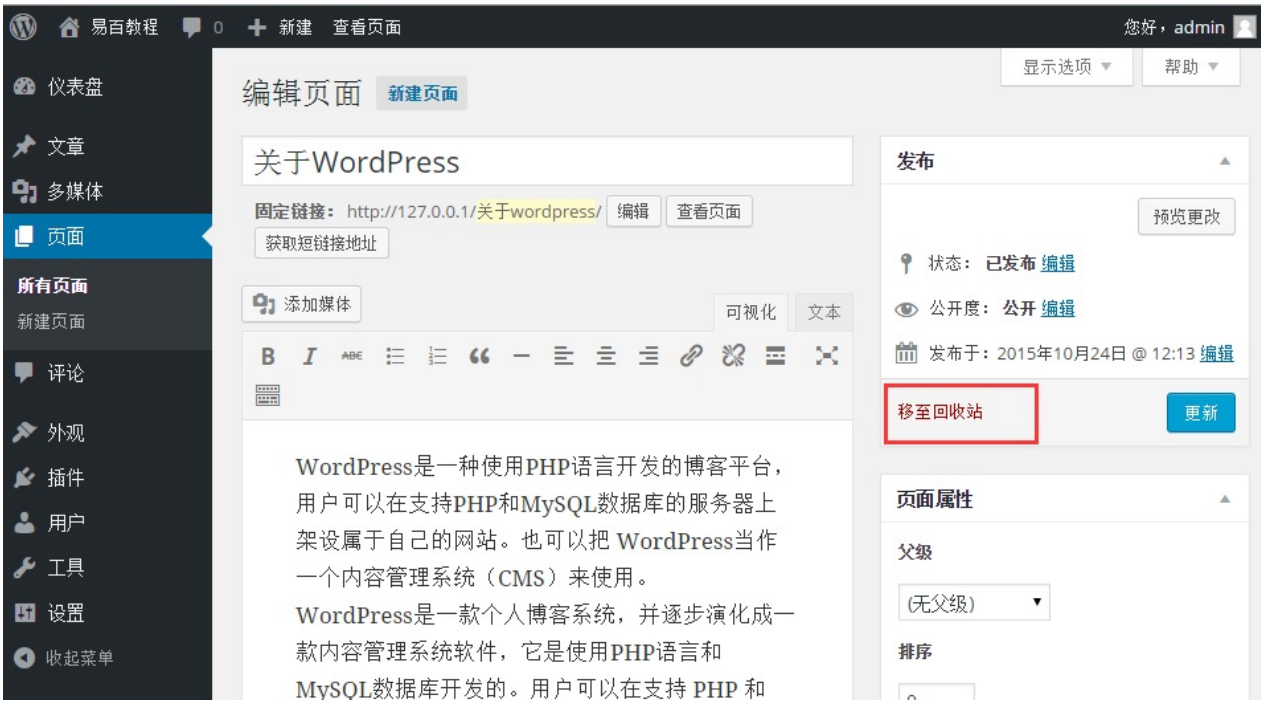
按照以下步骤删除WordPress的页面。

第**1**步 ：在WordPress中点击 页面 ->所有页面。

第**2**步 ：接下来，可以删除示例页面（示例页面是WordPress创建默认的）。当光标悬停在网页，几个选项显示在示例页面的下面。点击垃圾箱选项删除该文章页面。



或者可以直接删除页面，同时编辑或点击移动网页添加到垃圾桶，如下图所示：



第**3**步：接下来，以确认已删除的页面，查看您的网页列表。



Wordpress添加标签 - Wordpress教程

标签附加用于识别主要内容或文章信息的目的。它告诉访问者实际的文章是什么。如果所提到的标签正确，那么它有助于找到内容。

按照以下步骤在WordPress中添加标签。

第1步：在WordPress点击 文章-> 标签。

第2步：接着，将显示标签页。



以下是标签字段的详细信息。

- 名字: 输入标签的名称。
- 别名: 它被指定在标签URL。
- 描述: 添加您的标签的简要说明。当在标签上悬停它就会显示出来。

填写所有关于标签的信息之后，点击添加新标签按钮。

第2步：接下来，如下图所示，新创建的标签将得到显示在页面右侧。


添加新标签
名称
这将是它在站点上显示的名字。
别名
"别名"是在URL中使用的别称，它可以令URL更美观。通常使用小写，只能包含字母，数字和连字符（-）。
图像描述
描述只会在一部分主题中显示。
添加新标签
批量操作
应用
名称
图像描述
别名
总数
标签1
这是一个标签显示
tag1
0
名称
图像描述
别名
总数
批量操作
应用
标签可以有选择性地转换成分类目录，请使用标签到分类目录转换器。

Wordpress编辑标签 - Wordpress教程

在本章中，我们将学习在WordPress如何编辑标签。可以学习如何添加标签在WordPress添加标签章节 。以下是简单的步骤来在WordPress中编辑标签。

第1步：在WordPress点击 文章-> 标签。

第2步：接下来，你可以看到 tag1（tag1是在WordPress添加标签章节中创建）。当光标悬停在标签，有几个选项会显示在标签名称的下方。编辑标签有两种方式，即编辑和快速编辑

- 编辑：如下图所示，点击的标签部分中的编辑选项。



您可以编辑任何必填字段，然后点击更新按钮，如图下面的屏幕。



在这里，标签字段是和 WordPress添加标签章节是一样的。

- 快速编辑：如下图所示，点击的标签部分快速编辑选项。



在这里，你只能编辑如下图所示标签的名称和别名，然后点击更新标签按钮。



Wordpress删除标签 - Wordpress教程

在本章中，我们将学习如何在WordPress中删除标签。

可以学习如何添加标签在WordPress添加标签章节 。以下是简单的步骤来在WordPress中删除标签。

第1步 ：在WordPress点击 文章-> 标签。

第**2**步 ：接下来，你可以看到 tag1（tag1是在WordPress添加标签章节中创建）。当光标悬停在标签，有几个选项会显示在标签名称的下方。 Click on 点击删除标签部分，如下图所示。



当你点击删除，会得到一个弹出消息，要求您确认删除，如下图所示。



您可以点击 确定 按钮，并永久删除标签。

WordPress添加链接 - Wordpress教程

在本章中，我们将学习如何在WordPress的页面添加链接。链接是从一个资源到另一个的连接。添加链接到网页或博客帖子可以帮助您连接至其他页面。

以下是简单的步骤来在WordPress中添加链接。

第**1**步 ：在WordPress中点击 页面 ->所有页面，如下图所示。

第**2**步 ： 接下来，将得到显示在WordPress中创建的页面列表，如下图所示。选择任一页面，以在它里面添加链接。在这里，我们要在 关于易百教程 页面添加链接。



第**3**步 ：接下来，选择任何句子或单词，你想添加就可以了链接。在这里，我们将添加链接到



第4步： 接下来，当你点击插入/编辑链接符号然后弹出如下窗口(显示出来)。



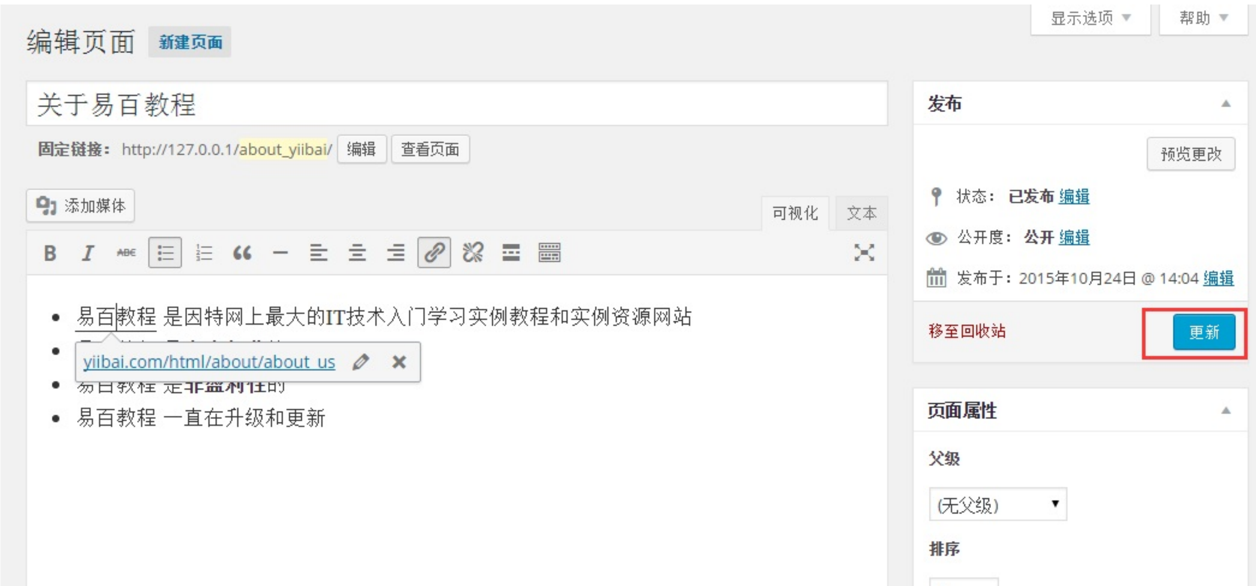
下面是插入/编辑链接的字段。

- URL: 输入你要链接的网址(目标网址)。
- 链接文本: 将要进入链接文本。
- 在新窗口或标签页打开链接: 它会打开网页链接到新的标签或窗口，根据需要勾选。
- 或链接到站点中的内容: 它通过从给定的列表中选择的页面链接到现有的内容页。点击或链接到站点中的内容，网页和帖子列表获取显示如下图所示。



从列表中选择特定页面或文章后，则该链接的URL字段创建如上图所示。点击添加链接。

第**5**步： 接着，如果对这些字悬停 易百教程 那么链路提示获得显示如下所示。



接下来，点击更新按钮来更新改变你的页面或文章。

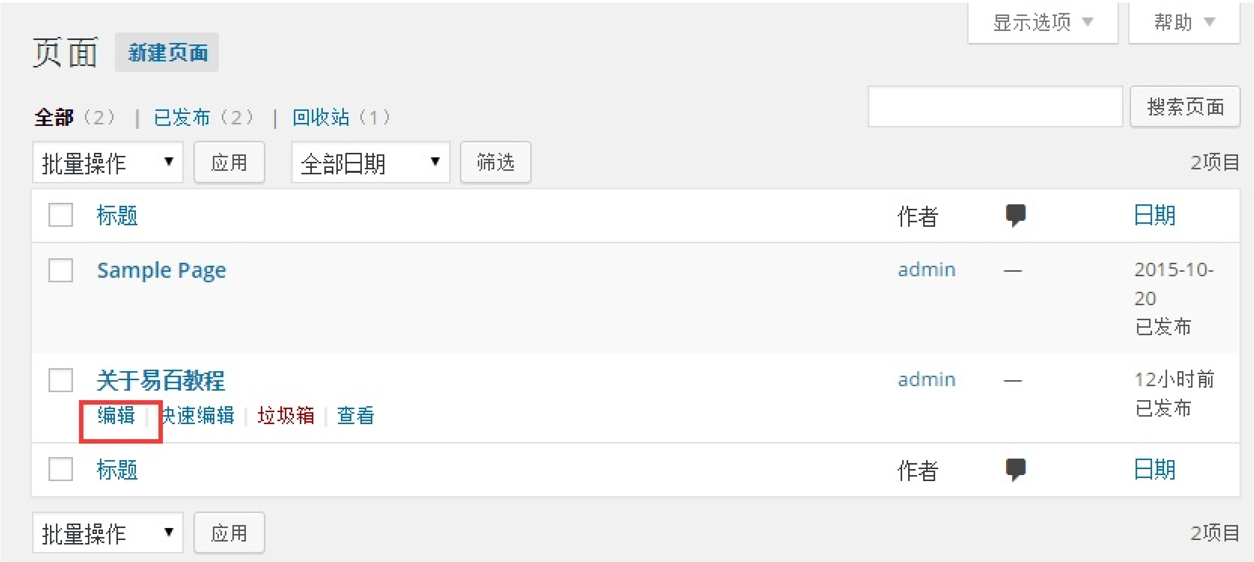
WordPress编辑链接 - Wordpress教程

在本章中，我们将研究如何在WordPress中编辑链接。你可以在WordPress添加链接这一章节中学习如何添加链接。

以下是简单的步骤来在WordPress编辑链接

第**1**步：点击页面 ->所有页面，如下图所示。

第**2**步：接下来，您可以查看页面列表。当光标悬停在 关于易百教程 页面上，几个选项被显示在 关于易百教程 的下面。点击编辑，如图所示。



第**3**步：接下来，将鼠标悬停在 关于易百教程（在 WordPress创建链接 这一章节中创建），并单击铅笔符号来编辑链接，如下图所示。

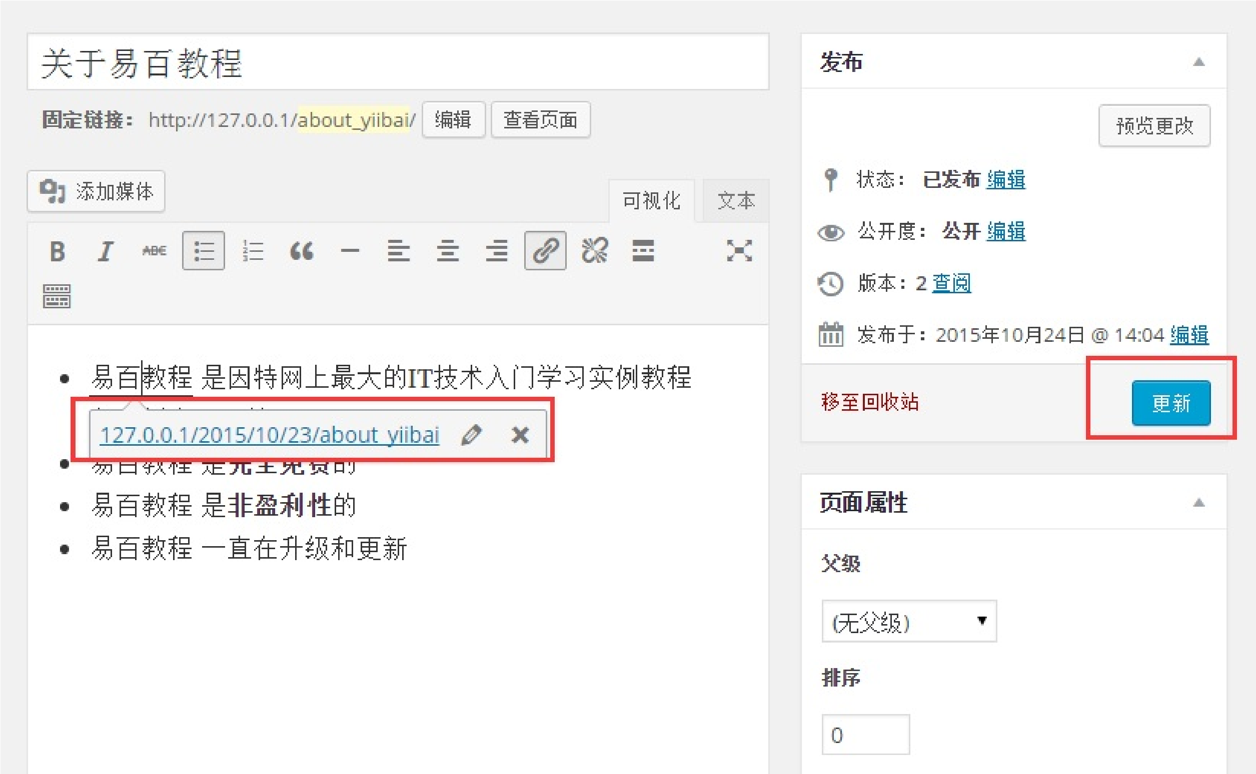


第4步：接下来，您可以从现有列表中选择页面中更改或修改链接。这里我们选择 关于易百教程 页面。



从列表中选择特定的页面或文章后，然后点击 更新 按钮。

第**5**步：接着，如果对 易百教程 悬停那么链接提示如下所示。



接下来，点击更新按钮来更新改变你的页面或文章。

Wordpress删除链接 - Wordpress教程

在本章中，我们将学习如何在WordPress中删除链接。你可以在WordPress添加链接这一章节中学习如何添加链接。

以下是简单的步骤来在WordPress删除链接

第1步：点击页面 ->所有页面，如下图所示。

第2步：接下来，可以查看页面列表。当光标悬停在 关于易百教程 页面上，几个选项显示 关于易百教程 的名称下面。点击编辑，如图所示。



第3步：接下来，选择您已经添加链接的话，即 易百教程（链接 易百教程 是WordPress添加链接中创建的），然后点击 删除链接 代码，如下图所示。



第**4**步：接着，如果对这些 易百教程 悬停，就没有现有工具提示URL。



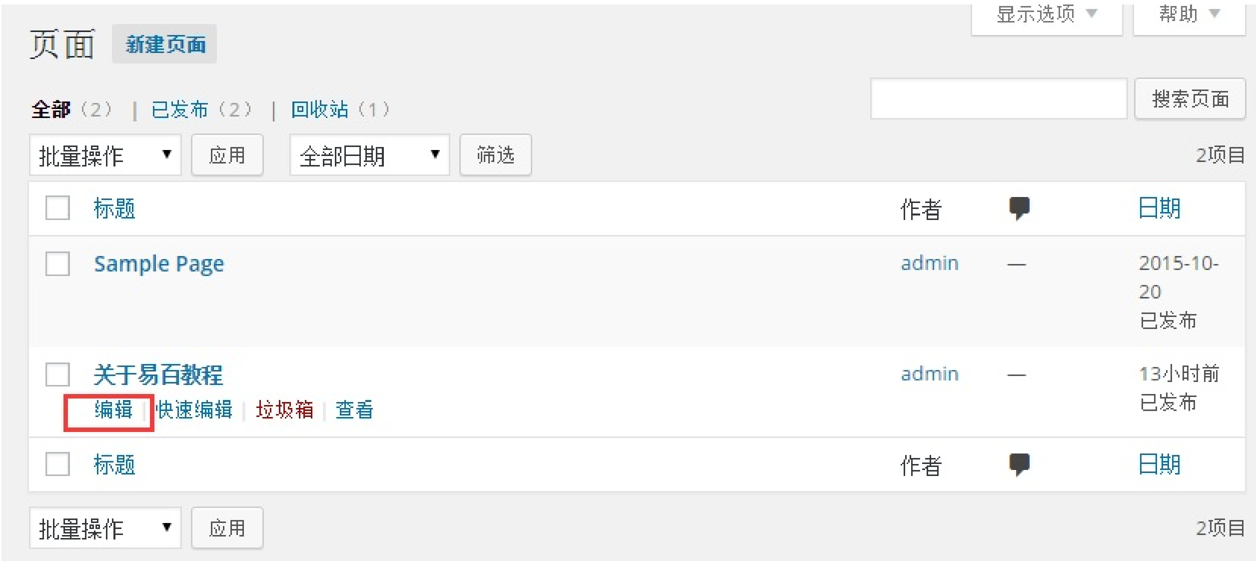
WordPress添加评论 - Wordpress教程

在本章中，我们将学习如何在WordPress中添加评论。添加评论可以让你的访客参与评论。评论由管理员审核发布以进一步讨论。

按照以下步骤将评论添加到您的博客文章。

步骤**(1)**：在WordPress中点击 页面 -> 所有页面。

步骤**(2)**：在WordPress中创建的页面将显示列表如下图所示。选择任何你想要添加注释的页面。在这里，我们要添加评论到 关于易百教程 页面。点击关于易百教程。



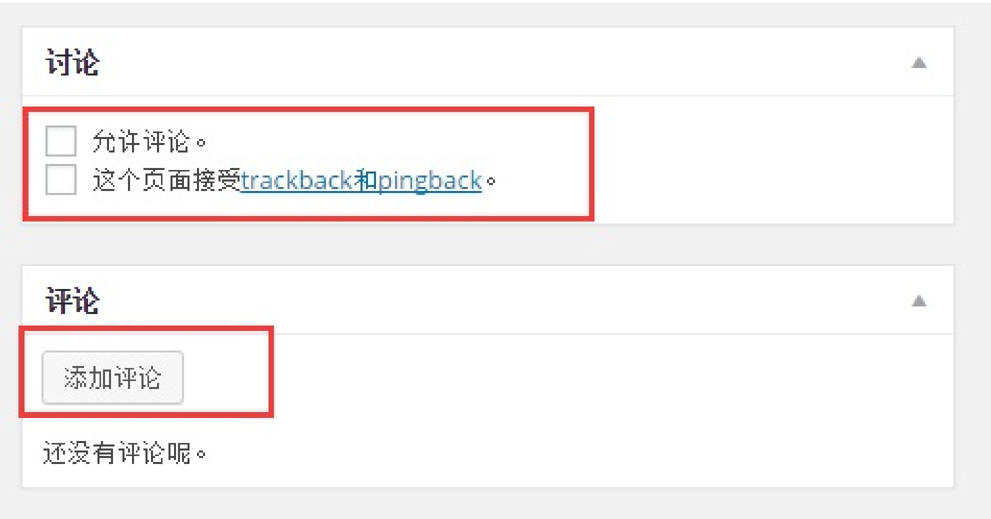
步骤**(3)**：接着，添加评论此页面上，点击出现在左上方的角落屏幕的选项。



步骤**(4)**：接下来，获取显示的屏幕选项的下拉列表。检查讨论和评论，如下图所示。



步骤(5)：接下来，可以在页面的底部查看讨论和评论框。



在评论部分，有两个选项存在：

- 允许评论: 让游客在你的博客文章和网页发表评论。
- 这个页面接受**trackback**和**pingback**。：让游客给ping和引用通告。

在评论部分中，您可以通过点击添加评论按钮添加评论。

步骤(6)：添加评论之后，点击更新按钮。


讨论
允许评论。
这个页面接受trackback和pingback。
评论
添加新评论
b
i
link
b-quote
del
ins
img
ul
ol
li
code
关闭标签
这是一个评论测试用例
取消
添加评论

Wordpress编辑评论 - Wordpress教程

在本章中，我们将学习如何在WordPress中编辑评论。编辑评论只能由管理员来完成。你可以学习如何在WordPress添加评论。

按照以下步骤是在WordPress中编辑评论。

步骤(1)：在WordPress中点击评论。



步骤（2）：接下来，你可以查看评论列表中的各个页面。选择任何评论对其进行编辑。点击编辑。



步骤（3）：接下来，编辑注释页面得到展示。您可以编辑注释，点击更新按钮。



在这里，可以编辑姓名，电子邮件，URL或评论框的评论内容。

WordPress查看插件 - Wordpress教程

在本章中，我们将研究如何在WordPress中查看插件。它可以帮助你启用和禁用WordPress插件。这增加了网站独特的功能。插件扩展和扩大WordPress的功能。

以下是简单的步骤来在在WordPress查看插件。

步骤**(1)**：管理员在 WordPress 中点击 插件 -> 已安装的插件，如下图所示。



步骤（**2**）：接下来，会看到您网站上的现有插件列表，如下图所示：



这里，将显示用插件和描述的表格。该插件的名称在插件栏定义，对插件的简要说明根据描述列来定义。

工具栏：

以下是在插件工具栏的选项：

- 启用**：**显示了网站上的活动（启用）插件。
- 未启用：显示已安装的插件，但处于非活动（未启用）状态的网站上。
- 更新可用： 显示如果有新版本可用，或者要求立即更新。

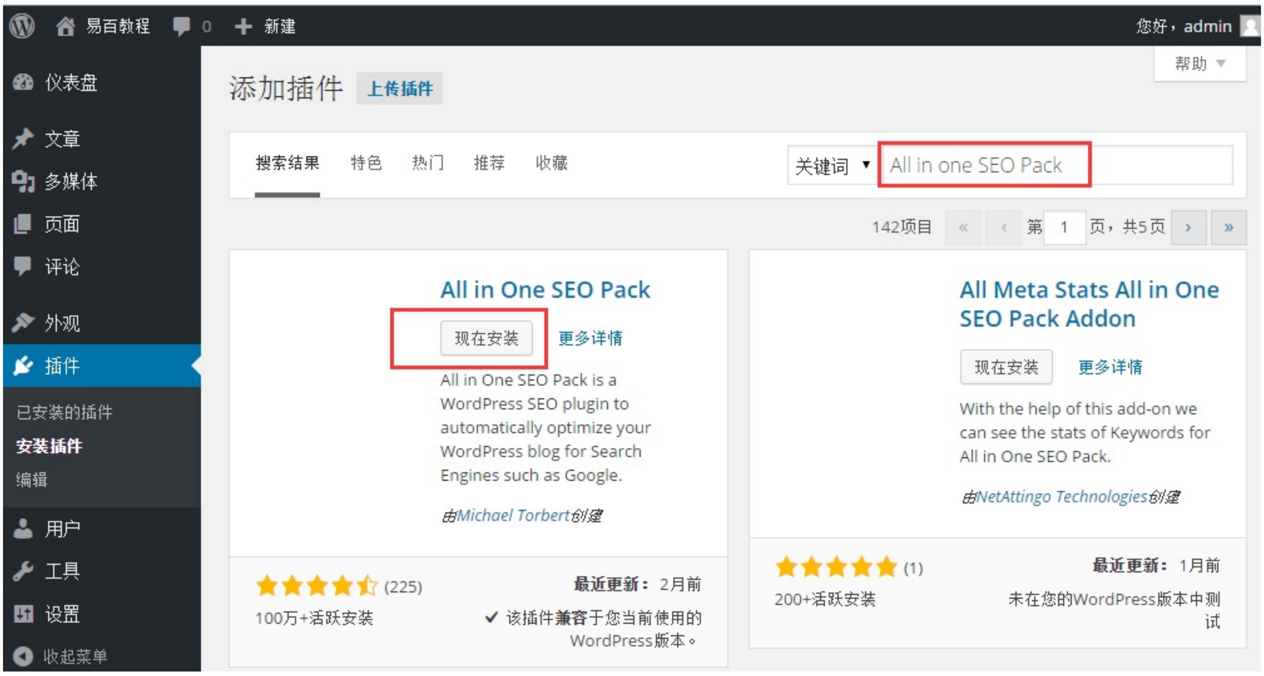
Wordpress安装插件 - Wordpress教程

在本章中，我们将学习如何在WordPress安装插件。在WordPress安装插件这是很容易的。所有的插件都可以免费下载;唯一的选择是一个插件必须在WordPress的目录中。

以下是在WordPress安装插件的简单的步骤。

步骤**(1)**：在WordPress中点击 插件 -> 安装插件。

步骤（**2**）：接下来，在搜索框中输入你需要的插件名称。插件这是相关的插件名称列表将得到显示，如图所示。



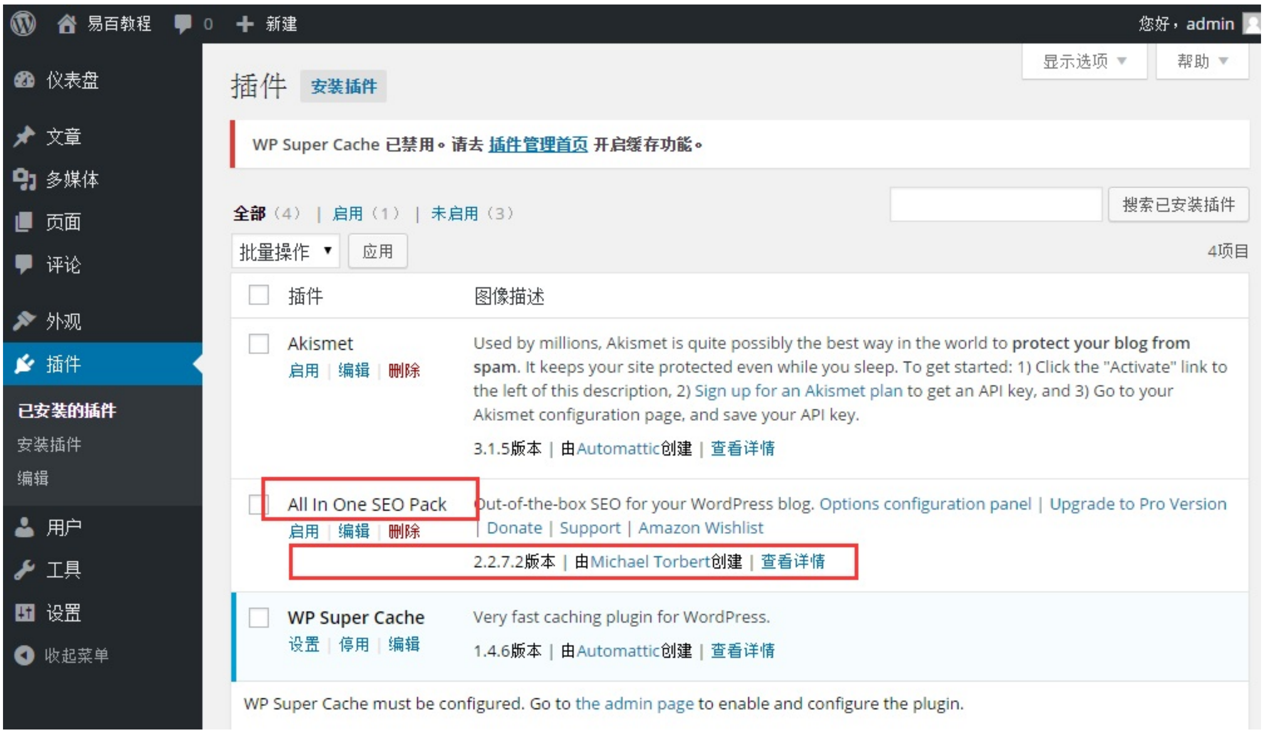
选择您想要使用的插件。在这里，我们搜索：All in one SEO Pack 插件, 这恰好是所述的第一个插件在屏幕上面显示。点击立即安装按钮，在您的网站上安装插件。

步骤**(3)**：接下来，插件开始下载并安装该插件。



点击启动插件激活插件在您的网站中，使用这个插件使您的工作更加容易。

步骤（4）：接下来，您将看到已安装插件的插件列表，如下图所示。



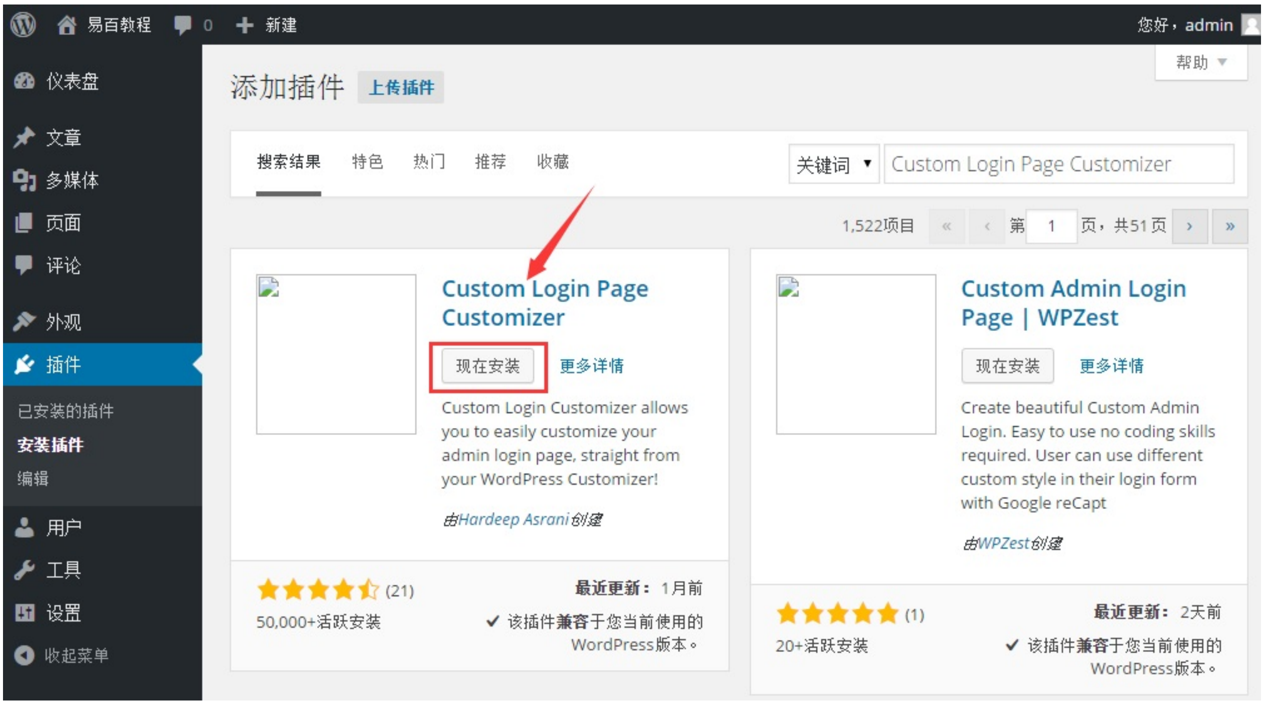
WordPress自定义插件 - Wordpress教程

在本章中，我们将学习如何在WordPress中自定义插件。它通常是多用户的网站。 在这里，我们了解学习如何在无需编写任何HTML或CSS自定义WordPress插件。 这种新的方法可以让你使用WordPress主题定制(无需编码技能)来定制你的登录页面。

以下是简单的步骤，在WordPress自定义插件。

步骤(1)：在WordPress中点击 插件 -> 安装插件。

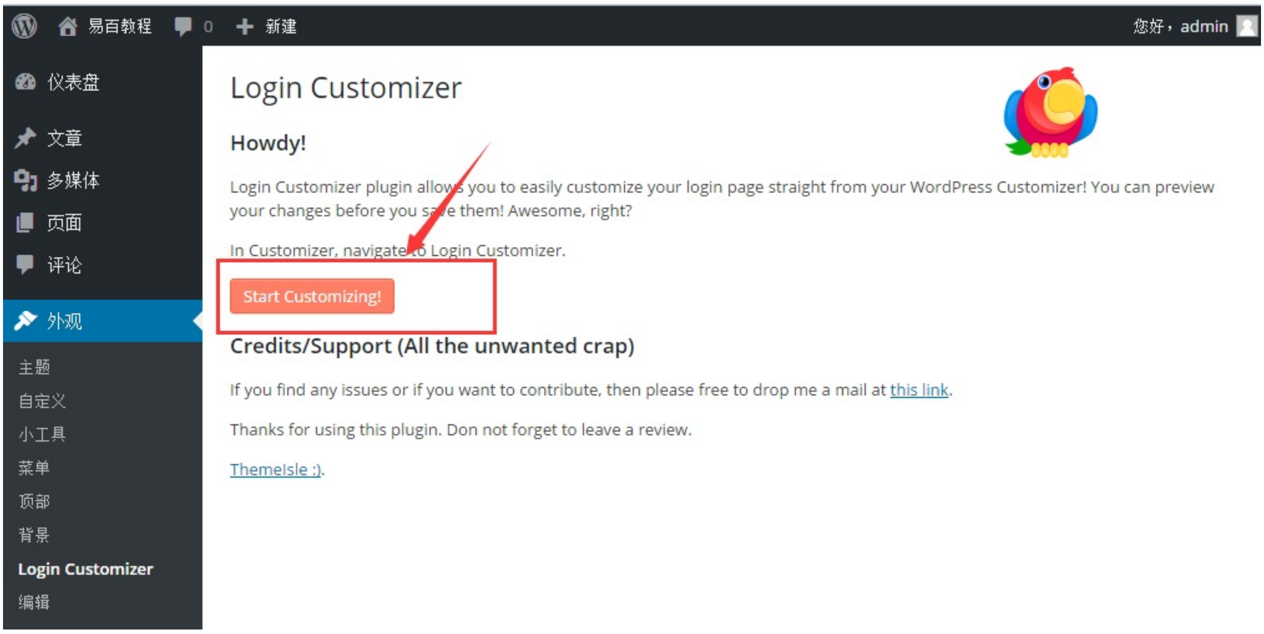
步骤（2）：安装并激活自定义登录页面自定义插件（**Custom Login Page Customizer**）。



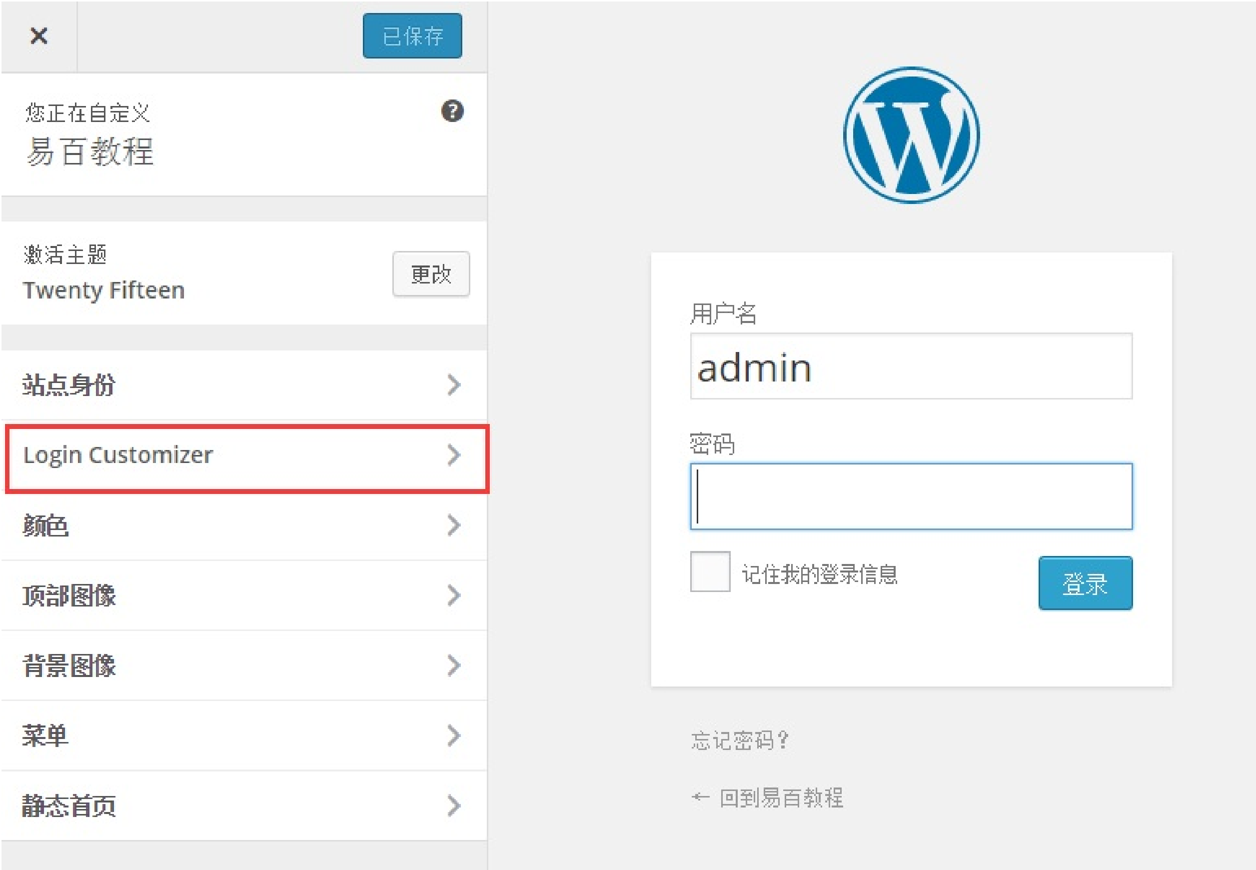
步骤（3）：接下来，在 WordPress 点击 外观 -> **Login Customizer(登录定制)**。



步骤（4）：点击开始按钮，进一步进行自定义。

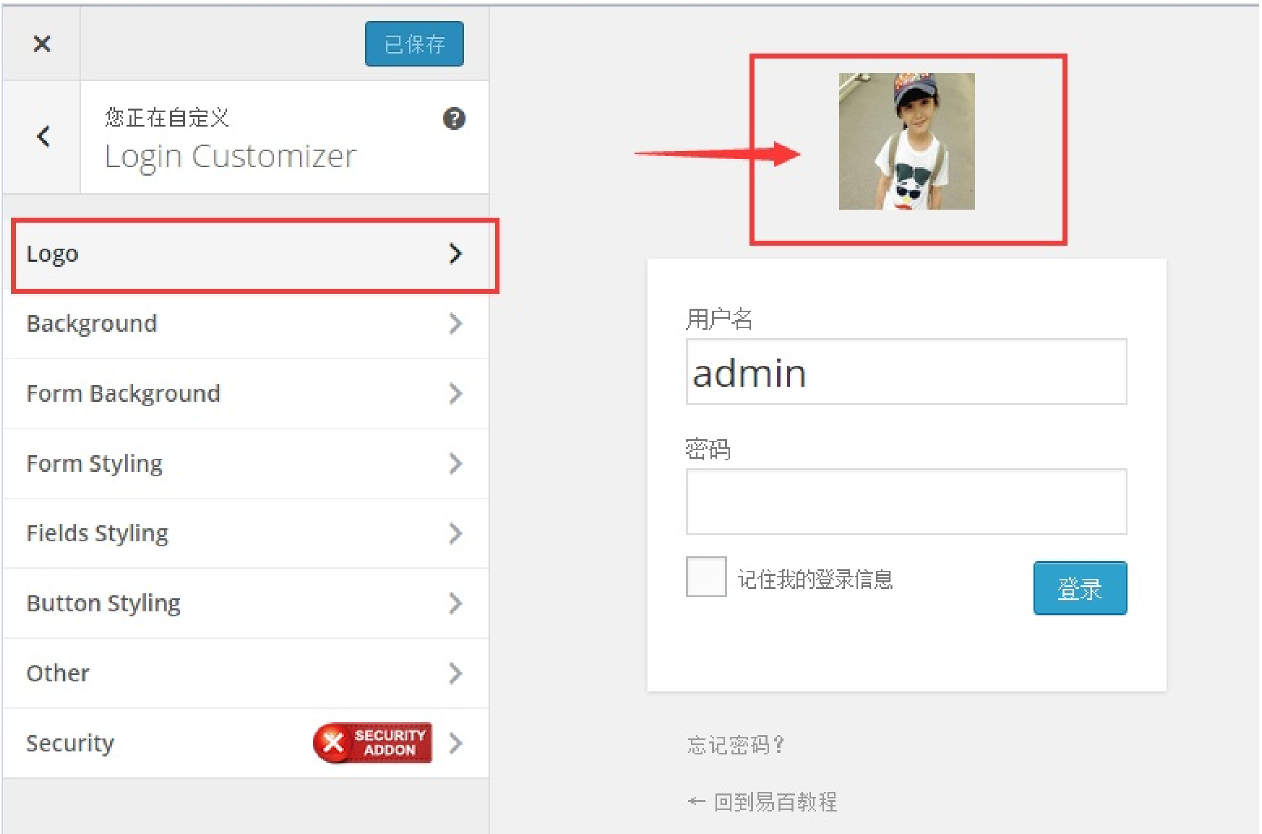


步骤（5）：接下来，它将启动内置 WordPress 主题定制。可以自定义你想要的主题方式。



在这里，点击侧面板中的新 Login Customizer 选项卡上。登录定制页面将得到显示。在登录定制页面，可以用同样的方式来定制你的登录页面，可以自定义WordPress的主题。

步骤（**6**）：接下来，您将看到自定义登录页面，作为一个演示示例，这里只是修改了登陆的Logo，如下图所示。



- 标志：选择上传标志来替换默认WordPress标志。
- 背景：添加背景图片，或者您可以选择您所选择的背景色。
- 表格背景：选择表单背景图片或颜色选择登录表单容器。

大多数选择在定制面板是透明的。可以检查出所有按您要求选择的定制调整设置登录页。点击保存并发布按钮。

Wordpress用户角色 - Wordpress教程

在本章中，我们将了解在WordPress的用户角色。 在WordPress的每个用户都有自己的作用。 角色就是给一个特定用户访问WordPress站点权限。这些角色只能由管理员进行分配。

这里是一些在WordPress中使用的几个角色：

- 管理员:管理员拥有所有的权限。管理员在WordPress的网站可以做任何事情，他可以创建更多管理员，邀请更多的用户，也可以删除它们。
- 编辑**:**编辑可以访问所有文章，页面，评论，分类，标签和链接。他们可以创建，发布，编辑或删除任文章位或网页。
- 作者:作者只能写文章或帖子，上传照片，编辑和发布自己的文章帖子。
- 贡献者**: **投稿者只能编写和编辑自己的文章帖子直到被发布。他们可以创建自己的文章和页面，但不能发布。他们无法上传图片或文件，但可以看到网站的状态。当他们想要发布的任何文章或帖子，必须先亲自通知管理员的撰稿人审查。当获得批准后投稿人可以不进行任何更改发表文章。
- 订阅者: 跟随只能读取，并在帖子发表评论。关注者可登录到帐户接收更新的内容。
- 查看者: 查看者只能查看自己的帖子;他们不能修改，但只能在帖子发表评论。

Yii2教程

Yii 是什么

===========Yii 是一个高性能，基于组件的 PHP 框架，用于快速开发现代 Web 应用程序。名字 Yii （读作 `易`）在中文里有“极致简单与不断演变”两重含义，也可看作 **Yes It Is**! 的缩写。

Yii 最适合做什么？

Yii 是一个通用的 Web 编程框架，即可以用于开发各种基于 PHP 的 Web 应用。因为基于组件的框架结构和设计精巧的缓存支持，Yii 特别适合开发大型应用，如门户网站、论坛、内容管理系统（CMS）、电子商务项目和 RESTful Web 服务等。

Yii 和其他框架相比呢？

- 和其他 PHP 框架类似，Yii 实现了 MVC（Model-View-Controller）设计模式并基于该模式组织代码。- Yii 的代码简洁优雅，这是 Yii 的编程哲学。它永远不会为了要迎合某个设计模式而对代码进行过度的设计。- Yii 是一个全栈框架，提供了大量久经考验，开箱即用的特性，例如：对关系型和 NoSQL 数据库都提供了查询生成器（QueryBuilders）和 ActiveRecord；RESTful API 的开发支持；多层缓存支持，等等。- Yii 非常易于扩展。你可以自定义或替换几乎任何一处核心代码。你还会受益于它坚实可靠的扩展架构，使用、再开发或再发布扩展。- 高性能始终是 Yii 的首要目标之一。Yii 不是一场独角戏，它由一个强大的开发者团队提供支持，也有一个庞大的专家社区，持续不断地对 Yii 的开发作出贡献。Yii 开发者团队始终对 Web 开发最新潮流和其他框架及项目中的最佳实践和特性保持密切关注，那些有意义的最佳实践及特性会被不定期的整合进核心框架中，并提供简单优雅的接口。

Yii 版本

Yii 当前有两个主要版本：1.1 和 2.0。 1.1 版是上代的老版本，现在处于维护状态。2.0 版是一个完全重写的版本，采用了最新的技术和协议，包括依赖包管理器（Composer）、PHP 代码规范（PSR）、命名空间、Traits（特质）等等。 2.0 版代表了最新一代框架，是未来几年中我们的主要开发版本。本指南主要基于 2.0 版编写。

系统要求和先决条件

Yii 2.0 需要 PHP 5.4.0 或以上版本支持。你可以通过运行任何 Yii 发行包中附带的系统要求检查器查看每个具体特性所需的 PHP 配置。使用 Yii 需要对面向对象编程（OOP）有基本了解，因为 Yii 是一个纯面向对象的框架。Yii 2.0 还使用了 PHP 的最新特性，例如 命名空间 和 Trait（特质）。理解这些概念将有助于你更快地掌握 Yii 2.0。

从 Yii 1.1升级到 Yii2 - Yii2教程

从 Yii 1.1 升级

2.0 版框架是完全重写的，在 1.1 和 2.0 两个版本之间存在相当多差异。因此从 1.1 版升级并不像小版本间的跨越那么简单，通过本指南你将会了解两个版本间主要的不同之处。如果你之前没有用过 Yii 1.1，可以跳过本章，直接从"入门篇"开始读起。请注意，Yii 2.0 引入了很多本章并没有涉及到的新功能。强烈建议你通读整部权威指南来了解所有新特性。这样有可能会发现一些以前你要自己开发的功能，而现在已经被包含在核心代码中了。

安装

Yii 2.0 完全拥抱 Composer，它是PHP中的一个依赖管理工具。核心框架以及扩展的安装都通过 Composer 来处理。想要了解更多如何安装 Yii 2.0 请参阅本指南的 安装 Yii 章节。如果你想创建新扩展，或者把你已有的 Yii 1.1 的扩展改写成兼容 2.0 的版本，你可以参考 创建扩展 章节。

PHP 需求

Yii 2.0 需要 PHP 5.4 或更高版本，该版本相对于 Yii 1.1 所需求的 PHP 5.2 而言有巨大的改进。因此在语言层面上有很多的值得注意的不同之处。下面是 PHP 层的主要变化汇总：- 命名空间- 匿名函数- 数组短语法 `[...元素...]` 用于取代 `array(...元素...)`- 视图文件中的短格式 echo 标签 `<?=`，自 PHP 5.4 起总会被识别并且合法，无论 short_open_tag 的设置是什么，可以安全使用。- SPL 类和接口- 延迟静态绑定- 日期和时间- Traits- intl Yii 2.0 使用 PHP 扩展 `intl` 来支持国际化的相关功能。

命名空间

Yii 2.0 里最明显的改动就数命名空间的使用了。几乎每一个核心类都引入了命名空间，比如 `yii\web\Request`。1.1 版用于类名前的字母 “C” 已经不再使用。当前的命名规范与目录结构相吻合。例如，`yii\web\Request` 就表明对应的类文件是 Yii 框架文件夹下的 `web/Request.php` 文件。(有了 Yii 的类自动加载器，你可以直接使用全部核心类而不需要显式包含具体文件。)

组件(Component)与对象(Object)

Yii 2.0 把 1.1 里的 `CComponent` 类拆分成了两个类：`yii\base\Object` 和 `yii\base\Component`。`yii\base\Object|Object` 类是一个轻量级的基类，你可以通过 getters 和 setters 来定义 object 的属性。`yii\base\Component|Component` 类继承自

`yii\base\Object|Object`，同时进一步支持 事件 和 行为。如果你不需要用到事件或行为，应该考虑使用 `yii\base\Object|Object` 类作为基类。这通常是表示基本数据结构的类。

对象的配置

`yii\base\Object|Object` 类引入了一种统一对象配置的方法。所有 `yii\base\Object|Object` 的子类都应该用以下方法声明它的构造方法(如果需要的话)，以正确配置它自身：

```
class MyClass extends \yii\base\Object
{
    public function __construct($param1, $param2, $config = [])
    {
        // ... 配置生效前的初始化过程

        parent::__construct($config);
    }

    public function init()
    {
        parent::init();

        // ...配置生效后的初始化过程
    }
}
```

在上面的例子里，构造方法的最后一个参数必须输入一个配置数组，包含一系列用于在方法结尾初始化相关属性的键值对。你可以重写 `yii\base\Object::init()|init()` 方法来执行一些需要在配置生效后进行的初始化工作。你可以通过遵循以下约定俗成的编码习惯，来使用配置数组创建并配置新的对象：

```
$object = Yii::createObject([
    'class' => 'MyClass',
    'property1' => 'abc',
    'property2' => 'cde',
], [$param1, $param2]);
```

更多有关配置的细节可以在配置章节找到。

事件(Event)

在 Yii 1 中，通常通过定义 `on` 开头的方法(例如 `onBeforeSave`)来创建事件。而在 Yii 2 中，你可以使用任意的事件名了。同时通过调用 `yii\base\Component::trigger()|trigger()` 方法来触发相关事件：

```
$event = new \yii\base\Event;
$component->trigger($eventName, $event);
```

要给事件附加一个事件句柄(Event Handler 或者叫事件处理器)，需要使用 `yii\base\Component::on()|on()` 方法：

```
$component->on($eventName, $handler);
// 要解除相关句柄，使用 off 方法：
// $component->off($eventName, $handler);
```

事件功能还有更多增强之处。要了解它们，请查看事件章节。

路径别名(Path Alias)

Yii 2.0 将路径别名的应用扩大至文件/目录路径和 URL。Yii 2.0 中路径别名必须以 `@` 符号开头，以区别于普通文件目录路径或 URL。例如 `@yii` 就是指向 Yii 安装目录的别名。绝大多数 Yii 核心代码都支持别名。例如 `yii\caching\FileCache::cachePath` 就同时支持路径别名或普通的目录地址。路径别名也和类的命名空间密切相关。建议给每一个根命名空间定义一个路径别名，从而无须额外配置，便可启动 Yii 的类自动加载机制。例如，因为有 `@yii` 指向 Yii 安装目录，那类似 `yii\web\Request` 的类就能被 Yii 自动加载。同理，若你用了一个第三方的类库，如 Zend Framework，你只需定义一个名为 `@Zend` 的路径别名指向该框架的安装目录。之后 Yii 就可以自动加载任意 Zend Framework 中的类了。更多路径别名信息请参阅路径别名章节。

视图(View)

Yii 2 中视图最明显的改动是视图内的特殊变量 `$this` 不再指向当前控制器或小部件，而是指向视图对象，它是 2.0 中引入的全新概念。视图对象为 `yii\web\View` 的实例，他代表了 MVC 模式中的视图部分。如果你想要在视图中访问一个控制器或小部件，可以使用 `$this->context` 。要在其他视图里渲染一个局部视图，使用 `$this->render()` ，而不是 `$this->renderPartial()` 。 `render()` 现在只返回渲染结果，而不是直接显示它，所以现在你必须显式地把它 **echo** 出来。像这样：

```
echo $this->render('_item', ['item' => $item]);
```

除了使用 PHP 作为主要的模板语言，Yii 2.0 也装备了两种流行模板引擎的官方支持：Smarty 和 Twig。过去的 Prado 模板引擎不再被支持。要使用这些模板引擎，你需要配置 `view` 应用组件，给它设置 `yii\base\View::$renderers|View::$renderers` 属性。具体请参阅模板引擎章节。

模型(Model)

Yii 2.0 使用 `yii\base\Model` 作为模型基类，类似于 1.1 的 `CModel` 。 `CFormModel` 被完全弃用了，现在要创建表单模型类，可以通过继承 `yii\base\Model` 类来实现。Yii 2.0 引进了名为 `yii\base\Model::scenarios()|scenarios()` 的新方法来声明支持的场景，并指明在哪个场景下某属性必须经过验证，可否被视为安全值等等。如：

```
public function scenarios()
{
    return [
        'backend' => ['email', 'role'],
        'frontend' => ['email', '!role'],
    ];
}
```

上面的代码声明了两个场景：`backend` 和 `frontend` 。对于 `backend` 场景，`email` 和 `role` 属性值都是安全的，且能进行批量赋值。对于 `frontend` 场景，`email` 能批量赋值而 `role` 不能。 `email` 和 `role` 都必须通过规则验证。 `yii\base\Model::rules()|rules()` 方法仍用于声明验证规则。注意，由于引入了 `yii\base\Model::scenarios()|scenarios()` ，现在已经没有 `unsafe` 验证器了。大多数情况下，如果 `yii\base\Model::rules()|rules()` 方法内已经完整地指定场景了，那就不必覆写 `yii\base\Model::scenarios()|scenarios()` ，也不必声明 `unsafe` 属性值。要了解更多有关模型的细节，请参考模型章节。

控制器(Controller)

Yii 2.0 使用 `yii\web\Controller` 作为控制器的基类，类似于 1.1 的 `CWebController` 。使用 `yii\base\Action` 作为操作类的基类。这些变化最明显的影响是，当你在写控制器操作的代码时，应该返回(return)要渲染的内容而不是输出(echo)它：

```
public function actionView($id)
{
    $model = \app\models\Post::findOne($id);
    if ($model) {
        return $this->render('view', ['model' => $model]);
    } else {
        throw new \yii\web\NotFoundHttpException;
    }
}
```

请查看 控制器(Controller) 章节了解有关控制器的更多细节。

小部件(Widget)

Yii 2.0 使用 `yii\base\Widget` 作为小部件基类，类似于 1.1 的 `CWidget` 。为了让框架获得更好的 IDE 支持，Yii 2.0 引进了一个调用小部件的新语法。包含 `yii\base\Widget::begin()|begin()` ， `yii\base\Widget::end()|end()` 和 `yii\base\Widget::widget()|widget()` 三个静态方法，用法如下：

```
use yii\widgets\Menu;
use yii\widgets\ActiveForm;

// 注意必须 **"echo"** 结果以显示内容
echo Menu::widget(['items' => $items]);

// 传递一个用于初始化对象属性的数组
$form = ActiveForm::begin([
    'options' => ['class' => 'form-horizontal'],
    'fieldConfig' => ['inputOptions' => ['class' => 'input-xlarge'`,
]);
... 表单输入栏都在这里 ...
ActiveForm::end();
```

更多细节请参阅小部件章节。

主题(Theme)

2.0 主题的运作方式跟以往完全不同了。它们现在基于路径映射机制，该机制会把一个源视图文件的路径映射到一个主题视图文件路径。举例来说，如果路径映射为 `['/web/views' => '/web/themes/basic']`，那么 `/web/views/site/index.php` 视图经过主题修饰的版本就会是 `/web/themes/basic/site/index.php`。也因此让主题现在可以应用在任何视图文件之上，甚至是渲染控制器上下文环境之外的视图文件或小部件。同样，`CThemeManager` 组件已经被移除了。取而代之的 `theme` 成为了 `view` 应用组件的一个可配置属性。更多细节请参考主题章节。

控制台应用(Console Application)

控制台应用现在如普通的 Web 应用程序一样，由控制器组成，控制台的控制器继承自 `yii\console\Controller`，类似于 1.1 的 `CConsoleCommand`。运行控制台命令使用 `yii <route>`，其中 `<route>` 代表控制器的路由(如 `sitemap/index`)。额外的匿名参数传递到对应的控制器操作方法，而有名的参数根据 `yii\console\Controller::options()` 的声明来解析。Yii 2.0 支持基于代码注释自动生成相关命令行帮助(help)信息。更多细节请参阅控制台命令章节。

国际化(I18N)

Yii 2.0 移除了原来内置的日期格式器和数字格式器，为了支持 PECL intl PHP module(PHP 的国际化扩展)的使用。消息翻译现在由 `i18n` 应用组件执行。该组件管理一系列消息源，允许使用基于消息类别的不同消息源。更多细节请参阅国际化(Internationalization)章节。

操作过滤器(Action Filters)

操作的过滤现在通过行为(behavior)来实现。要定义一个新的，自定义的过滤器，请继承 `yii\base\ActionFilter` 类。要使用一个过滤器，需要把过滤器类作为一个 `behavior` 绑定到控制器上。例如，要使用 `yii\filters\AccessControl` 过滤器，你需要在控制器内添加如下代码：

```
public function behaviors()
{
    return [
        'access' => [
            'class' => 'yii\filters\AccessControl',
            'rules' => [
                ['allow' => true, 'actions' => ['admin'], 'roles' => ['@'`,
            ],
        ],
    ];
}
```

更多细节请参考过滤器章节。

前端资源(Assets)

Yii 2.0 引入了一个新的概念，称为资源包(Asset Bundle)，以代替 1.1 的脚本包概念。一个资源包是一个目录下的资源文件集合(如 JavaScript 文件、CSS 文件、图片文件等)。每一个资源包被表示为一个类，该类继承自 `yii\web\AssetBundle` 。用 `yii\web\AssetBundle::register()` 方法注册一个资源包后，就使它的资源可被 Web 访问了，注册了资源包的页面会自动包含和引用资源包内指定的 JS 和 CSS 文件。更多细节请参阅 前端资源管理(Asset) 章节。

助手类(Helpers)

Yii 2.0 很多常用的静态助手类，包括： *`yii\helpers\Html`* `yii\helpers\ArrayHelper` *`yii\helpers\StringHelper`* `yii\helpers\FileHelper` * `yii\helpers\Json` 请参考助手一览 章节来了解更多。

表单

Yii 2.0 引进了表单栏(**field**)的概念，用来创建一个基于 `yii\widgets\ActiveForm` 的表单。一个表单栏是一个由标签、输入框、错误消息(可能还有提示文字)组成的容器，被表示为 `yii\widgets\ActiveField|ActiveField` 对象。使用表单栏建立表单的过程比以前更整洁利落：

```
<?php field($model, 'username') ?>
<?php  field($model, 'password')->passwordInput() ?>
```

请参考创建表单章节来了解更多细节。

查询生成器(Query Builder)

Yii 1.1 中，查询语句的生成分散在多个类中，包括 `CDbCommand`，`CDbCriteria` 以及 `CDbCommandBuilder`。Yii 2.0 以 `yii\db\Query|Query` 对象的形式表示一个数据库查询，这个对象使用 `yii\db\QueryBuilder|QueryBuilder` 在幕后生成 SQL 语句。例如：

```
$query = new \yii\db\Query();
$query->select('id, name')
      ->from('user')
      ->limit(10);

$command = $query->createCommand();
$sql = $command->sql;
$rows = $command->queryAll();
```

最重要的是，这些查询生成方法还可以和活动记录配合使用。请参考查询生成器(Query Builder)章节了解更多内容。

活动记录(Active Record)

Yii 2.0 的活动记录改动了很多。两个最显而易见的改动分别涉及查询语句的生成(query building)和关联查询的处理(relational query handling)。1.1 中的 `CDbCriteria` 类在 Yii 2 中被 `yii\db\ActiveQuery` 所替代。这个类是继承自 `yii\db\Query`，因此也继承了所有查询生成方法。开始拼装一个查询可以调用 `yii\db\ActiveRecord::find()` 方法进行：

```
// 检索所有 *活动的* 客户和订单，并以 ID 排序：
$customers = Customer::find()
    ->where(['status' => $active])
    ->orderBy('id')
    ->all();
```

要声明一个关联关系，只需简单地定义一个 getter 方法来返回一个 `yii\db\ActiveQuery|ActiveQuery` 对象。getter 方法定义的属性名(译者注：即 getOrders() 中的 orders)表示关联关系名。如，以下代码声明了一个名为 `orders` 的关系(1.1 中必须在 `relations()` 方法内声明关系)：

```
class Customer extends \yii\db\ActiveRecord
{
    public function getOrders()
    {
        return $this->hasMany('Order', ['customer_id' => 'id']);
    }
}
```

现在你就可以通过调用 `$customer->orders` 来访问关联表中某用户的订单了。你还可以用以下代码进行一场指定条件的实时关联查询：

```
$orders = $customer->getOrders()->andWhere('status=1')->all();
```

当贪婪加载一段关联关系时，Yii 2.0 和 1.1 的运作机理并不相同。具体来说，在 1.1 中使用一条 JOIN 语句同时查询主表和关联表记录。在 Yii 2.0 中会使用两个没有 JOIN 的 SQL 语句：第一条语句取回主表记录，第二条通过主表记录经主键筛选后查询关联表记录。当生成返回大量记录的查询时，可以链式书写 `yii\db\ActiveQuery::asArray()|asArray()` 方法，这样会以数组的形式返回查询结果，而不必返回 `yii\db\ActiveRecord|ActiveRecord` 对象，这能显著降低因大量记录读取所消耗的 CPU 时间和内存。如：

```
$customers = Customer::find()->asArray()->all();
```

另一个改变是你不能再通过公共数据定属性(Attribute)的默认值了。如果你需要这么做的话，可以在你的记录类的 `init` 方法中设置它们。

```
public function init()
{
    parent::init();
    $this->status = self::STATUS_NEW;
}
```

曾几何时，在 1.1 中重写一个活动记录类的构造方法(Constructor)会导致一些问题。它们不会在 2.0 中出现了。需要注意的是，如果你需要在构造方法中添加一些参数，恐怕必须重写 `yii\db\ActiveRecord::instantiate()` 方法。活动记录方面还有很多其他的变化与改进，请参考活动记录章节以了解更多细节。

用户及身份验证接口(IdentityInterface)

1.1 中的 `CWebUser` 类现在被 `yii\web\User` 所取代，随之 `CUserIdentity` 类也不在了。与之相对的，为达到相同目的，你可以实现 `yii\web\IdentityInterface` 接口，它使用起来更直观。在高级应用模版里提供了一个这样的一个例子。要了解更多细节请参考认证(Authentication)，授权(Authorization)以及高级应用模版 这三个章节。

URL 管理

Yii 2.0 的 URL 管理跟 1.1 中很像。一个主要的改进是现在的 URL 管理支持可选参数了。比如，如果你在 2.0 中定义了一个下面这样的规则，那么它可以同时匹配 `post/popular` 和 `post/1/popular` 两种 URL。而在 1.1 中为达成相同效果，必须要使用两条规则。

```
[
    'pattern' => 'post/<page:\d+>/<tag>',
    'route' => 'post/index',
    'defaults' => ['page' => 1],
]</tag></page:\d+>
```

请参考URL 解析和生成 章节，以了解更多细节。.

同时使用 Yii 1.1 和 2.x

如果你有一些遗留的 Yii 1.1 代码，需要跟 Yii 2.0 一起使用，可以参考 1.1 和 2.0 共用章节。

安装Yii2 - Yii2教程

安装 **Yii**==============你可以通过两种方式安装 Yii：使用 Composer 或下载一个归档文件。推荐使用前者，这样只需执行一条简单的命令就可以安装新的扩展或更新 Yii 了。> 注意：和 Yii 1 不同，以标准方式安装 Yii 2 时会同时下载并安装框架本身和一个应用程序的基本骨架。通过 **Composer** 安装 ----------------------如果还没有安装 Composer，你可以按 getcomposer.org 中的方法安装。在 Linux 和 Mac OS X 中可以运行如下命令：

```
curl -s http://getcomposer.org/installer | php
mv composer.phar /usr/local/bin/composer
```

在 Windows 中，你需要下载并运行 Composer-Setup.exe。如果遇到任何问题或者想更深入地学习 Composer，请参考 Composer 文档(英文)，Composer 中文。Composer 安装后，切换到一个可通过 Web 访问的目录，执行如下命令即可安装 Yii ：

```
composer create-project --prefer-dist yiisoft/yii2-app-basic basic
```

如上命令会将 Yii 安装在名为 `basic` 的目录中。> 技巧：如果你想安装 Yii 的最新开发版本，可以使用如下命令，它添加了一个 stability 选项(中文版):

```
composer create-project --prefer-dist --stability=dev yiisoft/yii2-app-basic basic
```

> 注意，Yii 的开发版(dev 版)不应该用于生产环境中，它可能会破坏运行中的代码。通过归档文件安装 -------------------------------通过归档文件安装 Yii 包括两个步骤：1. 从 yiiframework.com 下载归档文件。2. 将下载的文件解压缩到 Web 目录中。3. 修改 `config/web.php` 文件，给 `cookieValidationKey` 配置项添加一个密钥(若你通过 Composer 安装，则此步骤会自动完成)：
> `php // !!! 在下面插入一段密钥(若为空) - 以供 cookie validation 的需要 'cookieValidationKey' =`
> 其他安装方式 --------------------------上文介绍了两种安装 Yii 的方法，安装的同时也会创建一个立即可用的 Web 应用程序。对于小的项目或用于学习上手，这都是一个不错的起点。但是其他的安装方式也存在： 如果你只想安装核心框架，然后从零开始构建整个属于你自己的应用程序模版，可以参考从头构建自定义模版一节的介绍。 如果你要开发一个更复杂的应用，可以更好地适用于团队开发环境的，可以考虑安装高级应用模版。验证安装的结果 --------------------------安装完成后，就可以使用浏览器通过如下 URL 访问刚安装完的 Yii 应用了：

```
http://localhost/basic/web/index.php
```

这个 URL 假设你将 Yii 安装到了一个位于 Web 文档根目录下的 `basic` 目录中，且该 Web 服务器正运行在你自己的电脑上(`localhost`)。你可能需要将其调整为适应自己的安装环境。![Yii 安装成功]你应该可以在浏览器中看到如上所示的 “Congratulations!” 页面。如果没有，请通过以下任意一种方式，检查当前 PHP 环境是否满足 Yii 最基本需求： 通过浏览器访问 *URL* `http://localhost/basic/requirements.php` 执行如下命令：

```
http://localhost/basic/web/index.php
```

你需要配置好 PHP 安装环境，使其符合 Yii 的最小需求。主要是需要 PHP 5.4 以上版本。如果应用需要用到数据库，那还要安装 PDO PHP 扩展 和相应的数据库驱动(例如访问 MySQL 数据库所需的 `pdo_mysql`)。配置 **Web** 服务器---------------------->补充：如果你现在只是要试用 Yii 而不是将其部署到生产环境中，本小节可以跳过。通过上述方法安装的应用程序在 Windows，Max OS X，Linux 中的 Apache HTTP 服务器或 Nginx HTTP 服务器 上都可以直接运行。在生产环境的服务器上，你可能会想配置服务器让应用程序可以通过 URL `http://www.example.com/index.php` 访问而不是 `http://www.example.com/basic/web/index.php` 。这种配置需要将 Web 服务器的文档根目录指向 `basic/web` 目录。可能你还会想隐藏掉 URL 中的 `index.php` ，具体细节在 URL 解析和生成 一章中有介绍，你将学到如何配置 Apache 或 Nginx 服务器实现这些目标。>补充：将 `basic/web` 设置为文档根目录，可以防止终端用户访问 `basic/web` 相邻目录中的私有应用代码和敏感数据文件。禁止对其他目录的访问是一个不错的安全改进。>补充：如果你的应用程序将来要运行在共享虚拟主机环境中，没有修改其 Web 服务器配置的权限，你依然可以通过调整应用的结构来提升安全性。详情请参考共享主机环境 一章。### 推荐使用的 Apache 配置 在 Apache 的 `httpd.conf` 文件或在一个虚拟主机配置文件中使用如下配置。注意，你应该将 `path/to/basic/web` 替换为实际的 `basic/web` 目录。

```
# 设置文档根目录为 “basic/web”
DocumentRoot "path/to/basic/web"
 <directory basic="" path="" to="">RewriteEngine on

    # 如果请求的是真实存在的文件或目录，直接访问
    RewriteCond %{REQUEST_FILENAME} !-f
    RewriteCond %{REQUEST_FILENAME} !-d
    # 如果请求的不是真实文件或目录，分发请求至 index.php
    RewriteRule . index.php

    # ...其它设置...</directory>
```

推荐使用的 **Nginx** 配置 为了使用 **Nginx**，你应该已经将 **PHP** 安装为 **FPM SAPI** 了。使用如下 **Nginx** 配置，将 `path/to/basic/web` 替换为实际的 `basic/web` 目录， `mysite.local` 替换为实际的主机名以提供服务。

```
server {
    charset utf-8;
    client_max_body_size 128M;

    listen 80; ## 监听 ipv4 上的 80 端口
    #listen [::]:80 default_server ipv6only=on; ## 监听 ipv6 上的 80 端口

    server_name mysite.local;
    root        /path/to/basic/web;
    index       index.php;

    access_log  /path/to/basic/log/access.log main;
    error_log   /path/to/basic/log/error.log;

    location / {
        # 如果找不到真实存在的文件，把请求重定向给 index.php
        try_files $uri $uri/ /index.php?$args;
    }

    # 若取消下面这段的注释，可避免 Yii 接管不存在文件的处理过程(404)
    #location ~ .(js|css|png|jpg|gif|swf|ico|pdf|mov|fla|zip|rar)$ {
    #    try_files $uri =404;
    #}
    #error_page 404 /404.htmll;

    location ~ .php$ {
        include fastcgi.conf;
        fastcgi_pass   127.0.0.1:9000;
        #fastcgi_pass unix:/var/run/php5-fpm.sock;
    }

    location ~ /.(ht|svn|git) {
        deny all;
    }
}
```

使用该配置时，你还应该在 `php.ini` 文件中设置 `cgi.fix_pathinfo=0` ，能避免掉很多不必要的 `stat()` 系统调用。还要注意当运行一个 HTTPS 服务器时，需要添加 `fastcgi_param HTTPS on;` 一行，这样 Yii 才能正确地判断连接是否安全。

运行应用 - Yii2教程

运行应用

安装 Yii 后，就有了一个运行中的 Yii 应用，根据配置的不同，可以通过 `http://hostname/basic/web/index.php` 或 `http://hostname/index.php` 访问。本章节将介绍应用的内建功能，如何组织代码，以及一般情况下应用如何处理请求。 > 补充：为简单起见，在整个“入门”板块都假定你已经把 `basic/web` 设为 Web 服务器根目录并配置完毕，你访问应用的地址会是 `http://lostname/index.php` 或类似的。请按需调整 URL。

功能

一个安装完的基本应用包含四页：* 主页，当你访问 `http://hostname/index.php` 时显示, * “About” 页， * “Contact” 页， 显示一个联系表单，允许终端用户通过 Email 联系你， * “Login” 页， 显示一个登录表单，用来验证终端用户。试着用 “admin/admin” 登录，你可以看到当前是登录状态，已经可以“退出登录”了。 这些页面使用同一个头部和尾部。头部包含了一个可以在不同页面间切换的导航栏。 在浏览器底部可以看到一个工具栏。这是 Yii 提供的很有用的调试工具，可以记录并显示大量的调试信息，例如日志信息，响应状态，数据库查询等等。

应用结构

应用中最重要的目录和文件（假设应用根目录是 `basic`）：

```
basic/                  应用根目录
    composer.json       Composer 配置文件, 描述包信息
    config/             包含应用配置及其它配置
        console.php     控制台应用配置信息
        web.php         Web 应用配置信息
    commands/           包含控制台命令类
    controllers/        包含控制器类
    models/             包含模型类
    runtime/            包含 Yii 在运行时生成的文件，例如日志和缓存文件
    vendor/             包含已经安装的 Composer 包，包括 Yii 框架自身
    views/              包含视图文件
    web/                Web 应用根目录，包含 Web 入口文件
        assets/         包含 Yii 发布的资源文件（javascript 和 css）
        index.php       应用入口文件
    yii                 Yii 控制台命令执行脚本
```

一般来说，应用中的文件可被分为两类：在 `basic/web` 下的和在其它目录下的。前者可以直接通过 HTTP 访问（例如浏览器），后者不能也不应该被直接访问。 Yii 实现了模型-视图-控制器 (MVC)设计模式，这点在上述目录结构中也得以体现。 `models` 目录包含了所有[模型

类]， `views` 目录包含了所有[视图脚本]， `controllers` 目录包含了所有[控制器类]。 以下图表展示了一个应用的静态结构： [应用静态结构] 每个应用都有一个入口脚本 `web/index.php` ，这是整个应用中唯一可以访问的 PHP 脚本。入口脚本接受一个 Web 请求并创建[应用]实例去处理它。 [应用]在它的[组建]辅助下解析请求，并分派请求至 MVC 元素。 [视图]使用[小部件]去创建复杂和动态的用户界面。

请求生命周期

以下图表展示了一个应用如何处理请求： [请求生命周期] 1. 用户向[入口脚本] `web/index.php` 发起请求。 2. 入口脚本加载应用[配置]并创建一个[应用]实例去处理请求。 3. 应用通过[请求]组件解析请求的[路由]。 4. 应用创建一个[控制器]实例去处理请求。 5. 控制器创建一个[操作]实例并针对操作执行过滤器。 6. 如果任何一个过滤器返回失败，则操作退出。 7. 如果所有过滤器都通过，操作将被执行。 8. 操作会加载一个数据模型，或许是来自数据库。 9. 操作会渲染一个视图，把数据模型提供给它。 10. 渲染结果返回给[响应]组件。 11. 响应组件发送渲染结果给用户浏览器。

第一个程序HelloWord - Yii2教程

说声 Hello

本章节描述了如何在你的应用中创建一个新的 “Hello” 页面。为了做到这点，将会创建一个[操作]和一个[视图]：应用将会分派页面请求给操作 操作将会依次渲染视图呈现 “Hello” 给最终用户 贯穿整个章节，你将会掌握三件事： 1. 如何创建一个[操作]去响应请求， 2. 如何创建一个[视图]去构造响应内容， 3. 以及一个应用如何分派请求给[操作]。

创建操作

为了说 “Hello”，需要创建一个 `say` [操作]，从请求中接收 `message` 参数并显示给最终用户。如果请求没有提供 `message` 参数，操作将显示默认参数 “Hello”。 > 补充：[操作]是最终用户可以直接访问并执行的对象。操作被组织在[控制器]中。一个操作的执行结果就是最终用户收到的响应内容。 操作必须声明在[控制器]中。为了简单起见，你可以直接在 `SiteController` 控制器里声明 `say` 操作。这个控制器是由文件 `controllers/SiteController.php` 定义的。以下是一个操作的声明：

```
<?php
namespace app\controllers;
use yii\web\Controller;
class SiteController extends Controller
{
    // ...其它代码...
    public function actionSay($message = '你好')
    {
        return $this->render('say', ['message' => $message]);
    }
}
```

在上述 `SiteController` 代码中， `say` 操作被定义为 `actionSay` 方法。Yii 使用 `action` 前缀区分普通方法和操作。 `action` 前缀后面的名称被映射为操作的 ID。 涉及到给操作命名时，你应该理解 Yii 如何处理操作 ID。操作 ID 总是被以小写处理，如果一个操作 ID 由多个单词组成，单词之间将由连字符连接(如 `create-comment`)。操作 ID 映射为方法名时移除了连字符，将每个单词首字母大写，并加上 `action` 前缀。 例子：操作 ID `create-comment` 相当于方法名 `actionCreateComment` 。 上述代码中的操作方法接受一个参数 `$message` ，它的默认值是 `“Hello”`(就像你设置 PHP 中其它函数或方法的默认值一样)。当应用接收到请求并确定由 `say` 操作来响应请求时，应用将从请求的参数中寻找对应值传入进来。换句话说，如果请求包含一个 `message` 参数，它的值是 `“Goodybye”` ， 操作方法中的 `$message` 变量也将被填充为 `“Goodbye”` 。 在操作方法中， `yii\web\Controller::render()|render()` 被用来渲染一个名为 `say` 的视图文件。 `message` 参数也被传入视图，这样就可以在里面使用。操作方法会返回渲染结果。结果会被应用接收并显示给最终用户的浏览器(作为整页 HTML 的一部分)。

创建视图

[视图]是你用来生成响应内容的脚本。为了说 “Hello”，你需要创建一个 `say` 视图，以便显示从操作方法中传来的 `message` 参数。

```
<?php
use yii\helpers\Html;
?>
<?= Html::encode($message) ?>
```

`say` 视图应该存为 `views/site/say.php` 文件。当一个操作中调用了 `yii\web\Controller::render()|render()` 方法时，它将会按 `views/控制器 ID/视图名.php` 路径加载 PHP 文件。 注意以上代码， `message` 参数在输出之前被 `yii\helpers\Html::encode()|HTML-encoded` 方法处理过。这很有必要，当参数来自于最终用户时，参数中可能隐含的恶意 JavaScript 代码会导致跨站脚本(XSS)攻击。 当然了，你大概会在 `say` 视图里放入更多内容。内容可以由 HTML 标签，纯文本，甚至 PHP 语句组成。实际上 `say` 视图就是一个由 `yii\web\Controller::render()|render()` 执行的 PHP 脚本。视图脚本输出的内容将会作为响应结果返回给应用。应用将依次输出结果给最终用户。

尝试下

创建完操作和视图后，你就可以通过下面的 URL 访问新页面了：

```
http://hostname/index.php?r=site/say&message=Hello+World
```

[Hello World] 这个 URL 将会输出包含 “Hello World” 的页面，页面和应用里的其它页面使用同样的头部和尾部。 如果你省略 URL 中的 `message` 参数，将会看到页面只显示 “Hello”。这是因为 `message` 被作为一个参数传给 `actionSay()` 方法，当省略它时，参数将使用默认的 `“Hello”` 代替。 > 补充：新页面和其它页面使用同样的头部和尾部是因为 `yii\web\Controller::render()|render()` 方法会自动把 `say` 视图执行的结果嵌入称为[布局]的文件中，本例中是 `views/layouts/main.php` 。 上面 URL 中的参数 `r` 需要更多解释。它代表[路由]，是整个应用级的，指向特定操作的独立 ID。路由格式是 `控制器 ID/操作 ID` 。应用接受请求的时候会检查参数，使用控制器 ID 去确定哪个控制器应该被用来处理请求。然后相应控制器将使用操作 ID 去确定哪个操作方法将被用来做具体工作。上述例子中，路由 `site/say` 将被解析至 `SiteController` 控制器和其中的 `say` 操作。因此 `SiteController::actionSay()` 方法将被调用处理请求。 > 补充：与操作一样，一个应用中控制器同样有唯一的 ID。控制器 ID 和操作 ID 使用同样的命名规则。控制器的类名源自于控制器 ID，移除了连字符，每个单词首字母大写，并加上 `Controller` 后缀。例子：控制器 ID `post-comment` 相当于控制器类名 `PostCommentController` 。

总结

通过本章节你接触了 MVC 设计模式中的控制器和视图部分。创建了一个操作作为控制器的一部分去处理特定请求。然后又创建了一个视图去构造响应内容。在这个小例子中，没有模型调用，唯一涉及到数据的地方是 `message` 参数。 你同样学习了 Yii 路由的相关内容，它是用户请求与控制器操作之间的桥梁。 下一章，你将学习如何创建一个模型，以及添加一个包含 HTML 表单的页面。

使用表单 - Yii2教程

使用表单

本章节将介绍如何创建一个从用户那搜集数据的表单页。该页将显示一个包含 name 输入框和 email 输入框的表单。当搜集完这两部分信息后，页面将会显示用户输入的信息。

为了实现这个目标，除了创建一个[操作]和两个[视图]外，还需要创建一个[模型]。

贯穿整个小节，你将会学到：

- 创建一个模型代表用户通过表单输入的数据
- 声明规则去验证输入的数据
- 在视图中生成一个 HTML 表单

创建模型

模型类 `EntryForm` 代表从用户那请求的数据，该类如下所示并存储在 `models/EntryForm.php` 文件中。请参考[类自动加载]章节获取更多关于类命名约定的介绍。

```
<?php
namespace app\models;
use yii\base\Model;
class EntryForm extends Model
{
    public $name;
    public $email;

    public function rules()
    {
        return [
            `'name', 'email'], 'required'],
            ['email', 'email'],
        ];
    }
}
```

该类继承自 `yii\base\Model`，Yii 提供的一个基类，通常用来表示数据。

> 补充：`yii\base\Model` 被用于普通模型类的父类并与数据表无关。`yii\db\ActiveRecord` 通常是普通模型类的父类但与数据表有关联(译者注：`yii\db\ActiveRecord` 类其实也是继承自 `yii\base\Model`，增加了数据库处理)。

`EntryForm` 类包含 `name` 和 `email` 两个公共成员，用来储存用户输入的数据。它还包含一个名为 `rules()` 的方法，用来返回数据验证规则的集合。上面声明的验证规则表示：

- `name` 和 `email` 值都是必须的

- `mail` 的值必须满足 email 地址验证

如果你有一个从用户那搜集数据的 `EntryForm` 对象，你可以调用它的 `yii\base\Model::validate()|validate()` 方法触发数据验证。如果有数据验证失败，将把 `yii\base\Model::hasErrors|hasErrors` 属性设为 ture，想要知道具体发生什么错误就调用 `yii\base\Model::getErrors|getErrors` 。

```
<?php
$model = new EntryForm();
$model->name = 'Qiang';
$model->email = 'bad';
if ($model->validate()) {
    // 验证成功！
} else {
    // 失败！
    // 使用 $model->getErrors() 获取错误详情
}
```

创建操作

下面你得在 `site` 控制器中创建一个 `entry` 操作用于新建的模型。操作的创建和使用已经在说一声你好小节中解释了。

```
<?php
namespace app\controllers;
use Yii;
use yii\web\Controller;
use app\models\EntryForm;
class SiteController extends Controller
{
    // ...其它代码...

    public function actionEntry()
    {
        $model = new EntryForm;

        if ($model->load(Yii::$app->request->post()) && $model->validate()) {
            // 验证 $model 收到的数据

            // 做些有意义的事 ...

            return $this->render('entry-confirm', ['model' => $model]);
        } else {
            // 无论是初始化显示还是数据验证错误
            return $this->render('entry', ['model' => $model]);
        }
    }
}
```

该操作首先创建了一个 `EntryForm` 对象。然后尝试从 `$_POST` 搜集用户提交的数据，由 Yii 的 `yii\web\Request::post()` 方法负责搜集。如果模型被成功填充数据(也就是说用户已经提交了 HTML 表单)，操作将调用 `yii\base\Model::validate()|validate()` 去确保用户提交的是有效数据。

> 补充：表达式 `Yii::$app` 代表[应用]实例，它是一个全局可访问的单例。同时它也是一个[服务定位器]，能提供 `request`，`response`，`db` 等等特定功能的组件。在上面的代码里就是使用 `request` 组件来访问应用实例收到的 `$_POST` 数据。

用户提交表单后，操作将会渲染一个名为 `entry-confirm` 的视图去确认用户输入的数据。如果没填表单就提交，或数据包含错误(译者：如 email 格式不对)，`entry` 视图将会渲染输出，连同表单一起输出的还有验证错误的详细信息。

> 注意：在这个简单例子里我们只是呈现了有效数据的确认页面。实践中你应该考虑使用 `yii\web\Controller::refresh()|refresh()` 或 `yii\web\Controller::redirect()|redirect()` 去避免表单重复提交问题。

创建视图

最后创建两个视图文件 `entry-confirm` 和 `entry`。他们会被刚才创建的 `entry` 操作渲染。

`entry-confirm` 视图简单地显示提交的 name 和 email 数据。视图文件保存在 `views/site/entry-confirm.php`。

```
<?php
use yii\helpers\Html;
?>
<p>You have entered the following information:</p>

<ul>
    <li><label>Name</label>: <?= Html::encode($model->name) ?></li>
    <li><label>Email</label>: <?= Html::encode($model->email) ?></li>
</ul>
```

`entry` 视图显示一个 HTML 表单。视图文件保存在 `views/site/entry.php`。

```
<?php
use yii\helpers\Html;
use yii\widgets\ActiveForm;
?>
<?php $form = ActiveForm::begin(); ?>
    <?= $form->field($model, 'name') ?>
    <?= $form->field($model, 'email') ?>
    <div class="form-group">
        <?= Html::submitButton('Submit', ['class' => 'btn btn-primary']) ?>
    </div>
<?php ActiveForm::end(); ?>
```

视图使用了一个功能强大的小部件 `yii\widgets\ActiveForm|ActiveForm` 去生成 HTML 表单。其中的 `begin()` 和 `end()` 分别用来渲染表单的开始和关闭标签。在这两个方法之间使用了 `yii\widgets\ActiveForm::field()|field()` 方法去创建输入框。第一个输入框用于 “name”，第二个输入框用于 “email”。之后使用 `yii\helpers\Html::submitButton()` 方法生成提交按钮。

尝试下

用浏览器访问下面的 URL 看它能否工作：

```
http://hostname/index.php?r=site/entry
```

你会看到一个包含两个输入框的表单的页面。每个输入框的前面都有一个标签指明应该输入的数据类型。如果什么都不填就点击提交按钮，或填入格式不正确的 email 地址，将会看到在对应的输入框下显示错误信息。 [验证错误的表单] 输入有效的 name 和 email 信息并提交后，将会看到一个显示你所提交数据的确认页面。 [输入数据的确认页]

效果说明

你可能会好奇 HTML 表单暗地里是如何工作的呢，看起来它可以为每个输入框显示文字标签，而当你没输入正确的信息时又不需要刷新页面就能给出错误提示，似乎有些神奇。

是的，其实数据首先由客户端 JavaScript 脚本验证，然后才会提交给服务器通过 PHP 验证。 `yii\widgets\ActiveForm` 足够智能到把你在 `EntryForm` 模型中声明的验证规则转化成客户端 JavaScript 脚本去执行验证。如果用户浏览器禁用了 JavaScript， 服务器端仍然会像 `actionEntry()` 方法里这样验证一遍数据。这保证了任何情况下用户提交的数据都是有效的。

> 警告：客户端验证是提高用户体验的手段。无论它是否正常启用，服务端验证则都是必须的，请不要忽略它。

输入框的文字标签是 `field()` 方法生成的，内容就是模型中该数据的属性名。例如模型中的 `name` 属性生成的标签就是 `Name` 。

你可以在视图中自定义标签：

```
<?= $form->field($model, 'name')->label('自定义 Name') ?>
<?= $form->field($model, 'email')->label('自定义 Email') ?>
```

> 补充：Yii 提供了相当多类似的小部件去帮你生成复杂且动态的视图。在后面你还会了解到自己写小部件是多么简单。你可能会把自己的很多视图代码转化成小部件以提高重用，加快开发效率。

总结

本章节指南中你接触了 MVC 设计模式的每个部分。学到了如何创建一个模型代表用户数据并验证它的有效性。

你还学到了如何从用户那获取数据并在浏览器上回显给用户。这本来是开发应用的过程中比较耗时的任务，好在 Yii 提供了强大的小部件让它变得如此简单。

下一章你将学习如何使用数据库，几乎每个应用都需要数据库。

Yii2使用数据库 - Yii2教程

使用数据库

本章节将介绍如何如何创建一个从数据表 `country` 中获取国家数据并显示出来的页面。为了实现这个目标，你将会配置一个数据库连接，创建一个活动记录类，并且创建一个操作及一个视图。

贯穿整个章节，你将会学到：

- 配置一个数据库连接
- 定义一个活动记录类
- 使用活动记录从数据库中查询数据
- 以分页方式在视图中显示数据

请注意，为了掌握本章你应该具备最基本的数据库知识和使用经验。尤其是应该知道如何创建数据库，如何通过数据库终端执行 SQL 语句。

准备数据库

首先创建一个名为 `yii2basic` 的数据库，应用将从这个数据库中获取数据。你可以创建 SQLite，MySQL，PostregSQL，MSSQL 或 Oracle 数据库，Yii 内置多种数据库支持。简单起见后面的内容将以 MySQL 为例做演示。

然后在数据库中创建一个名为 `country` 的表并插入简单的数据。可以执行下面的语句：

```
CREATE TABLE `country` (
  `code` CHAR(2) NOT NULL PRIMARY KEY,
  `name` CHAR(52) NOT NULL,
  `population` INT(11) NOT NULL DEFAULT '0'
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

INSERT INTO `Country` VALUES ('AU','Australia',18886000);
INSERT INTO `Country` VALUES ('BR','Brazil',170115000);
INSERT INTO `Country` VALUES ('CA','Canada',1147000);
INSERT INTO `Country` VALUES ('CN','China',1277558000);
INSERT INTO `Country` VALUES ('DE','Germany',82164700);
INSERT INTO `Country` VALUES ('FR','France',59225700);
INSERT INTO `Country` VALUES ('GB','United Kingdom',59623400);
INSERT INTO `Country` VALUES ('IN','India',1013662000);
INSERT INTO `Country` VALUES ('RU','Russia',146934000);
INSERT INTO `Country` VALUES ('US','United States',278357000);
```

于是便有了一个名为 `yii2basic` 的数据库，在这个数据库中有一个包含三个字段的数据表 `country`，表中有十行数据。

配置数据库连接

开始之前，请确保你已经安装了 PHP PDO 扩展和你所使用的数据库的 PDO 驱动(例如 MySQL 的 `pdo_mysql`)。对于使用关系型数据库来讲，这是基本要求。

驱动和扩展安装可用后，打开 `config/db.php` 修改里面的配置参数对应你的数据库配置。该文件默认包含这些内容：

```
<?php

return [
    'class' => 'yiidbConnection',
    'dsn' => 'mysql:host=localhost;dbname=yii2basic',
    'username' => 'root',
    'password' => '',
    'charset' => 'utf8',
];
```

`config/db/php` 是一个典型的基于文件的配置工具。这个文件配置了数据库连接 `yiidbConnection` 的创建和初始化参数，应用的 SQL 查询正是基于这个数据库。

上面配置的数据库连接可以在应用中通过 `Yii::$app->db` 访问。

> 补充： `config/db.php` 将被包含在应用配置文件 `config/web.php` 中，后者指定了整个应用如何初始化。请参考配置章节了解更多信息。

创建活动记录

创建一个继承自活动记录类的类 `Country` ，把它放在 `models/Country.php` ，去表示和获取 `country` 表的数据。

```
<?php

namespace appmodels;

use yiidbActiveRecord;

class Country extends ActiveRecord
{
}
```

这个 `Country` 类继承自 `yiidbActiveRecord` 。你不用在里面写任何代码。只需要像现在这样，Yii 就能根据类名去猜测对应的数据表名。

> 补充：如果类名和数据表名不能直接对应，可以重写 `yiidbActiveRecord::tableName()|tableName()` 方法去显式指定相关表名。

使用 `Country` 类可以很容易地操作 `country` 表数据，就像这段代码：

```
use appmodelsCountry;

// 获取 country 表的所有行并以 name 排序
$countries = Country::find()->orderBy('name')->all();

// 获取主键为 “US” 的行
$country = Country::findOne('US');

// 输出 “United States”
echo $country->name;

// 修改 name 为 “U.S.A.” 并在数据库中保存更改
$country->name = 'U.S.A.';
$country->save();
```

> 补充：活动记录是面向对象、功能强大的访问和操作数据库数据的方式。你可以在活动记录章节了解更多信息。除此之外你还可以使用另一种更原生的称做数据访问对象的方法操作数据库数据。

创建操作

为了向最终用户显示国家数据，你需要创建一个操作。相比之前小节掌握的在 `site` 控制器中创建操作，在这里为所有和国家有关的数据新建一个控制器更加合理。新控制器名为 `CountryController` ，并在其中创建一个 `index` 操作，如下：

```
<?php

namespace appcontrollers;

use yiiwebController;
use yiidataPagination;
use appmodelsCountry;

class CountryController extends Controller
{
    public function actionIndex()
    {
        $query = Country::find();

        $pagination = new Pagination([
            'defaultPageSize' => 5,
            'totalCount' => $query->count(),
        ]);

        $countries = $query->orderBy('name')
            ->offset($pagination->offset)
            ->limit($pagination->limit)
            ->all();

        return $this->render('index', [
            'countries' => $countries,
            'pagination' => $pagination,
        ]);
    }
}
```

把上面的代码保存在 `controllers/CountryController.php` 。

`index` 操作调用了活动记录 `Country::find()` 方法，去生成查询语句并从 `country` 表中取回所有数据。为了限定每个请求所返回的国家数量，查询在 `yiidataPagination` 对象的帮助下进行分页。 `Pagination` 对象的使命主要有两点：

- 为 SQL 查询语句设置 `offset` 和 `limit` 从句，确保每个请求只需返回一页数据(本例中每页是 5 行)。
- 在视图中显示一个由页码列表组成的分页器，这点将在后面的段落中解释。

在代码末尾， `index` 操作渲染一个名为 `index` 的视图，并传递国家数据和分页信息进去。

创建视图

在 `views` 目录下先创建一个名为 `country` 的子目录。这个目录存储所有由 `country` 控制器渲染的视图。在 `views/country` 目录下创建一个名为 `index.php` 的视图文件，内容如下：

```
<?php
use yiihelpersHtml;
use yiiwidgetsLinkPager;
?>
<h1>Countries</h1>
<ul>
<?php foreach ($countries as $country): ?>
    <li>
        <?= Html::encode("{$country->name} ({$country->code})") ?>:
        <?= $country->population ?>
    </li>
<?php endforeach; ?>
</ul>

<?= LinkPager::widget(['pagination' => $pagination]) ?>
```

这个视图包含两部分用以显示国家数据。第一部分遍历国家数据并以无序 HTML 列表渲染出来。第二部分使用 `yiiwidgetsLinkPager` 去渲染从操作中传来的分页信息。小部件 `LinkPager` 显示一个分页按钮的列表。点击任何一个按钮都会跳转到对应的分页。

尝试下

浏览器访问下面的 URL 看看能否工作：

```
http://hostname/index.php?r=country/index
```

首先你会看到显示着五个国家的列表页面。在国家下面，你还会看到一个包含四个按钮的分页器。如果你点击按钮 “2”，将会跳转到显示另外五个国家的页面，也就是第二页记录。如果观察仔细点你还会看到浏览器的 URL 变成了：

```
http://hostname/index.php?r=country/index&page=2
```

在这个场景里，`yiidataPagination|Pagination` 提供了为数据结果集分页的所有功能：

- 首先 `yiidataPagination|Pagination` 把 SELECT 的子查询 `LIMIT 5 OFFSET 0` 数据表示成第一页。因此开头的五条数据会被取出并显示。
- 然后小部件 `yiiwidgetsLinkPager|LinkPager` 使用 `yiidataPagination::createUrl()|Pagination::createUrl()` 方法生成的 URL 去渲染翻页按钮。URL 中包含必要的参数 `page` 才能查询不同的页面编号。
- 如果你点击按钮 “2”，将会发起一个路由为 `country/index` 的新请求。`yiidataPagination|Pagination` 接收到 URL 中的 `page` 参数把当前的页码设为 2。新的数据库请求将会以 `LIMIT 5 OFFSET 5` 查询并显示。

总结

本章节中你学到了如何使用数据库。你还学到了如何取出并使用 `yiidataPagination` 和 `yiiwidgetsLinkPager` 显示数据。

下一章中你会学到如何使用 Yii 中强大的代码生成器 Gii，去帮助你实现一些常用的功能需求，例如增查改删(CRUD)数据表中的数据。事实上你之前所写的代码全部都可以由 Gii 自动生成。

使用Gii生成代码 - Yii2教程

使用 Gii 生成代码

本章节将介绍如何使用 Gii 去自动生成 Web 站点常用功能的代码。使用 Gii 生成代码非常简单，只要按照 Gii 页面上的介绍输入正确的信息即可。

贯穿本章节，你将会学到：

- 在你的应用中开启 Gii
- 使用 Gii 去生成活动记录类
- 使用 Gii 去生成数据表操作的增查改删(CRUD)代码
- 自定义 Gii 生成的代码

开始 Gii

Gii 是 Yii 中的一个模块。可以通过配置应用的 `yiiaseApplication::modules|modules` 属性开启它。通常来讲在 `config/web.php` 文件中会有以下配置代码：

```
$config = [ ... ];

if (YII_ENV_DEV) {
    $config['bootstrap'][] = 'gii';
    $config['modules']['gii'] = 'yiigiiModule';
}
```

这段配置的意思是如果当前是开发环境，应用会包含 `gii` 模块，模块类是 `yiigiiModule` 。

如果你检查应用的入口脚本 `web/index.php` ，将看到这行代码将 `YII_ENV_DEV` 设为 true：

```
defined('YII_ENV') or define('YII_ENV', 'dev');
```

代码设置应用处于开发模式下，按照上面的配置会打开 Gii 模块。你可以直接通过 URL 访问 Gii：

```
http://hostname/index.php?r=gii
```

生成活动记录类

选择 “Model Generator” (点击 Gii 首页的链接)去生成活动记录类。并像这样填写表单：

- Table Name: `country`
- Model Class: `Country`

然后点击 “Preview” 按钮。你会看到 `models/Country.php` 被列在将要生成的文件列表中。可以点击文件名预览内容。

如果你已经创建过同样的文件，使用 Gii 可以覆写它，点击文件名旁边的 `diff` 能查看现有文件与将要生成的文件的内容区别。

想要覆写已存在文件，选中 “overwrite” 下的复选框然后点击 “Generator”。如果是新文件，只点击 “Generator” 就好。

接下来你会看到一个包含已生成文件的说明页面。如果生成过程中覆写过文件，还会有一条信息说明代码是重新生成覆盖的。

生成 CRUD 代码

CRUD 代表增，查，改，删操作，这是绝大多数 Web 站点常用的数据处理方式。选择 Gii 中的 “CRUD Generator” (点击 Gii 首页的链接)去创建 CRUD 功能。之前的 “country” 例子需要像这样填写表单：

- Model Class: `appmodelsCountry`
- Search Model Class: `appmodelsCountrySearch`
- Controller Class: `appcontrollersCountryController`

然后点击 “Preview” 按钮。你会看到下述将要生成的文件列表。

`NEED THE IMAGE HERE / 等待官方补充图片`

如果你之前创建过 `controllers/CountryController.php` 和 `views/country/index.php` 文件(在指南的使用数据库小节)，选中 “overwrite” 下的复选框覆写它们(之前的文件没能全部支持 CRUD)。

尝试下

用浏览器访问下面的 URL 查看生成代码的运行：

```
http://hostname/index.php?r=country/index
```

可以看到一个栅格显示着从数据表中获取的国家数据。支持在列头对数据进行排序，输入筛选条件进行筛选。

可以浏览详情，编辑，或删除栅格中的每个国家。还可以点击栅格上方的 “Create Country” 按钮通过表单创建新国家。

下面列出由 Gii 生成的文件，以便你研习功能和实现，或修改它们。

- 控制器：`controllers/CountryController.php`
- 模型：`models/Country.php` 和 `models/CountrySearch.php`
- 视图：`views/country/*.php`

> 补充：Gii 被设计成高度可定制和可扩展的代码生成工具。使用它可以大幅提高应用开发速度。请参考 Gii 小节了解更多内容。

总结

本章学习了如何使用 Gii 去生成为数据表中数据实现完整 CRUD 功能的代码。

组件（Component） - Yii2教程

组件(Component)

组件是 Yii 应用的主要基石。是 `yiiaseComponent` 类或其子类的实例。三个用以区分它和其它类的主要功能有：

- 属性(Property)
- 事件(Event)
- 行为(Behavior)

或单独使用，或彼此配合，这些功能的应用让 Yii 的类变得更加灵活和易用。以小部件 `yiijuiDatePicker|日期选择器` 来举例，这是个方便你在 视图 中生成一个交互式日期选择器的 UI 组件：

```
1.  use yiijuiDatePicker;

3.  echo DatePicker::widget([
4.  'language' =&gt; 'zh-CN',
5.  'name'  =&gt; 'country',
6.  'clientOptions' =&gt; [
7.  'dateFormat' =&gt; 'yy-mm-dd',
8.  ],
9.  ]);
```

这个小部件继承自 `yiiaseComponent`，它的各项属性改写起来会很容易。

正是因为组件功能的强大，他们比常规的对象(Object)稍微重量级一点，因为他们要使用额外的内存和 CPU 时间来处理 事件 和 行为 。如果你不需要这两项功能，可以继承 `yiiaseObject` 而不是 `yiiaseComponent`。这样组件可以像普通 PHP 对象一样高效，同时还支持属性(Property)功能。

当继承 `yiiaseComponent` 或 `yiiaseObject` 时，推荐你使用如下的编码风格：

- 若你需要重写构造方法(Constructor)，传入 $config 作为构造器方法最后一个参数，然后把它传递给父类的构造方法。
- 永远在你重写的构造方法结尾处调用一下父类的构造方法。
- 如果你重写了 `yiiaseObject::init()` 方法，请确保你在 init 方法的开头处调用了父类的 init 方法。

例子如下：

```
1.  namespace yiicomponentsMyClass;

3.  use yiiaseObject;

5.  class MyClass extends Object
6.  {
7.  public $prop1;
8.  public $prop2;

10.  public function __construct($param1, $param2, $config = [])
11.  {
12.  // ... 配置生效前的初始化过程

14.  parent::__construct($config);
15.  }

17.  public function init()
18.  {
19.  parent::init();

21.  // ... 配置生效后的初始化过程
22.  }
23.  }
```

另外，为了让组件可以在创建实例时能被正确配置，请遵照以下操作流程：

```
1.  $component = new MyClass(1, 2, ['prop1' =&gt; 3, 'prop2' =&gt; 4]);
2.  // 方法二：
3.  $component = Yii::createObject([
4.  'class' =&gt; MyClass::className(),
5.  'prop1' =&gt; 3,
6.  'prop2' =&gt; 4,
7.  ], [1, 2]);
```

> 补充：尽管调用 `Yii::createObject()` 的方法看起来更加复杂，但这主要因为它更加灵活强大，它是基于依赖注入容器实现的。

`yiiaseObject` 类执行时的生命周期如下：

1. 构造方法内的预初始化过程。你可以在这儿给各属性设置缺省值。
2. 通过 $config 配置对象。配置的过程可能会覆盖掉先前在构造方法内设置的默认值。
3. 在 `yiiaseObject::init()|init()` 方法内进行初始化后的收尾工作。你可以通过重写此方法，进行一些良品检验，属性的初始化之类的工作。
4. 对象方法调用。

前三步都是在对象的构造方法内发生的。这意味着一旦你获得了一个对象实例，那么它就已经初始化就绪可供使用。

属性（Property） - Yii2教程

属性(Property)

在 PHP 中，类的成员变量也被称为属性**(properties)**。它们是类定义的一部分，用来表现一个实例的状态(也就是区分类的不同实例)。在具体实践中，常常会想用一个稍微特殊些的方法实现属性的读写。例如，要对 `label` 属性执行 trim 操作，可以用以下代码实现：

```
1.  $object-&gt;label = trim($label);
```

上述代码的缺点是只要修改 `label` 属性就必须再次调用 `trim()` 函数。若将来需要用其它方式处理 `label` 属性，比如首字母大写，就不得不修改所有给 `label` 属性赋值的代码。这种代码的重复会导致 bug，这种实践显然需要尽可能避免。

为解决该问题，Yii 引入了一个名为 `yii\base\Object` 的基类，它支持基于类内的 **getter** 和 **setter**(读取器和设定器)方法来定义属性。如果某类需要支持这个特性，只需要继承 `yii\base\Object` 或其子类即可。

> 补充：几乎每个 Yii 框架的核心类都继承自 `yii\base\Object` 或其子类。这意味着只要在核心类中见到 getter 或 setter 方法，就可以像调用属性一样调用它。

getter 方法是名称以 `get` 开头的方法，而 setter 方法名以 `set` 开头。方法名中 `get` 或 `set` 后面的部分就定义了该属性的名字。如下面代码所示，getter 方法 `getLabel()` 和 setter 方法 `setLabel()` 操作的是 `label` 属性，：

```
1.  namespace app\\components;

3.  use yii\\base\\Object;

5.  class Foo extend Object
6.  {
7.  private $_label;

9.  public function getLabel()
10.  {
11.  return $this-&gt;_label;
12.  }

14.  public function setLabel($value)
15.  {
16.  $this-&gt;_label = trim($value);
17.  }
18.  }
```

(详细解释：getter 和 setter 方法创建了一个名为 `label` 的属性，在这个例子里，它指向一个私有的内部属性 `_label` 。)

getter/setter 定义的属性用法与类成员变量一样。两者主要的区别是：当这种属性被读取时，对应的 getter 方法将被调用；而当属性被赋值时，对应的 setter 方法就调用。如：

```
1.  // 等效于 $label = $object-&gt;getLabel();
2.  $label = $object-&gt;label;

4.  // 等效于 $object-&gt;setLabel('abc');
5.  $object-&gt;label = 'abc';
```

只定义了 getter 没有 setter 的属性是只读属性。尝试赋值给这样的属性将导致 `yii\base\InvalidCallException|InvalidCallException` (无效调用)异常。类似的，只有 setter 方法而没有 getter 方法定义的属性是只写属性，尝试读取这种属性也会触发异常。使用只写属性的情况几乎没有。

通过 getter 和 setter 定义的属性也有一些特殊规则和限制：

- 这类属性的名字是不区分大小写的。如，$object->label 和 $object->Label 是同一个属性。因为 PHP 方法名是不区分大小写的。
- 如果此类属性名和类成员变量相同，以后者为准。例如，假设以上 Foo 类有个 label 成员变量，然后给$object->label = 'abc' 赋值，将赋给成员变量而不是 setter setLabel() 方法。
- 这类属性不支持可见性(访问限制)。定义属性的 getter 和 setter 方法是 public、protected 还是 private 对属性的可见性没有任何影响。
- 这类属性的 getter 和 setter 方法只能定义为非静态的，若定义为静态方法(static)则不会以相同方式处理。

回到开头提到的问题，与其处处要调用 `trim()` 函数，现在我们只需在 setter `setLabel()` 方法内调用一次。如果 label 首字母变成大写的新要求来了，我们只需要修改 `setLabel()` 方法，而无须接触任何其它代码。

事件 - Yii2教程

事件

事件可以将自定义代码“注入”到现有代码中的特定执行点。附加自定义代码到某个事件，当这个事件被触发时，这些代码就会自动执行。例如，邮件程序对象成功发出消息时可触发 `messageSent` 事件。如想追踪成功发送的消息，可以附加相应追踪代码到 `messageSent` 事件。

Yii 引入了名为 `yii\base\Component` 的基类以支持事件。如果一个类需要触发事件就应该继承 `yii\base\Component` 或其子类。

事件处理器(Event Handlers)

事件处理器是一个PHP 回调函数，当它所附加到的事件被触发时它就会执行。可以使用以下回调函数之一：

- 字符串形式指定的 PHP 全局函数，如 'trim' ；
- 对象名和方法名数组形式指定的对象方法，如 [$object, $method] ；
- 类名和方法名数组形式指定的静态类方法，如 [$class, $method] ；
- 匿名函数，如 function ($event) { ... } 。

事件处理器的格式是：

```
1.  function ($event) {
2.  // $event 是 yii\\base\\Event 或其子类的对象
3.  }
```

通过 `$event` 参数，事件处理器就获得了以下有关事件的信息：

- `yii\base\Event::name|event name` ：事件名
- `yii\base\Event::sender|event sender` ：调用 trigger() 方法的对象
- `yii\base\Event::data|custom data` ：附加事件处理器时传入的数据，默认为空，后文详述

附加事件处理器

调用 `yii\base\Component::on()` 方法来附加处理器到事件上。如：

```
1.  $foo = new Foo;

3.  // 处理器是全局函数
4.  $foo-&gt;on(Foo::EVENT_HELLO, 'function_name');

6.  // 处理器是对象方法
7.  $foo-&gt;on(Foo::EVENT_HELLO, [$object, 'methodName']);

9.  // 处理器是静态类方法
10.  $foo-&gt;on(Foo::EVENT_HELLO, ['app\\components\\Bar', 'methodName']);

12.  // 处理器是匿名函数
13.  $foo-&gt;on(Foo::EVENT_HELLO, function ($event) {
14.  //事件处理逻辑
15.  });
```

附加事件处理器时可以提供额外数据作为 `yii\base\Component::on()` 方法的第三个参数。数据在事件被触发和处理器被调用时能被处理器使用。如：

```
1.  // 当事件被触发时以下代码显示 "abc"
2.  // 因为 $event-&gt;data 包括被传递到 "on" 方法的数据
3.  $foo-&gt;on(Foo::EVENT_HELLO, function ($event) {
4.  echo $event-&gt;data;
5.  }, 'abc');
```

时间处理器顺序

可以附加一个或多个处理器到一个事件。当事件被触发，已附加的处理器将按附加次序依次调用。如果某个处理器需要停止其后的处理器调用，可以设置 `$event` 参数的 `yii\base\Event::handled` 属性为真，如下：

```
1.  $foo-&gt;on(Foo::EVENT_HELLO, function ($event) {
2.  $event-&gt;handled = true;
3.  });
```

默认新附加的事件处理器排在已存在处理器队列的最后。因此，这个处理器将在事件被触发时最后一个调用。在处理器队列最前面插入新处理器将使该处理器最先调用，可以传递第四个参数 `$append` 为假并调用 `yii\base\Component::on()` 方法实现：

``php $foo->on(Foo::EVENT_HELLO, function ($event) { // 这个处理器将被插入到处理器队列的第一位... }, $data, false);

```
 触发事件
----------

事件通过调用 `yii\base\Component::trigger()` 方法触发，此方法须传递**事件名**，还可以传递一个事件对

```php
namespace app\components;

use yii\base\Component;
use yii\base\Event;

class Foo extends Component
{
    const EVENT_HELLO = 'hello';

    public function bar()
    {
        $this->trigger(self::EVENT_HELLO);
    }
}
```

以上代码当调用 `bar()` ，它将触发名为 `hello` 的事件。

> 提示：推荐使用类常量来表示事件名。上例中，常量 `EVENT_HELLO` 用来表示 `hello` 。这有两个好处。第一，它可以防止拼写错误并支持 IDE 的自动完成。第二，只要简单检查常量声明就能了解一个类支持哪些事件。

有时想要在触发事件时同时传递一些额外信息到事件处理器。例如，邮件程序要传递消息信息到 `messageSent` 事件的处理器以便处理器了解哪些消息被发送了。为此，可以提供一个事件对象作为 `yii\base\Component::trigger()` 方法的第二个参数。这个事件对象必须是 `yii\base\Event` 类或其子类的实例。如：

```
1.  namespace app\\components;

3.  use yii\\base\\Component;
4.  use yii\\base\\Event;

6.  class MessageEvent extends Event
7.  {
8.  public $message;
9.  }

11.  class Mailer extends Component
12.  {
13.  const EVENT_MESSAGE_SENT = 'messageSent';

15.  public function send($message)
16.  {
17.  // ...发送 $message 的逻辑...

19.  $event = new MessageEvent;
20.  $event->message = $message;
21.  $this->trigger(self::EVENT_MESSAGE_SENT, $event);
22.  }
23.  }
```

当 `yii\base\Component::trigger()` 方法被调用时，它将调用所有附加到命名事件(trigger 方法第一个参数)的事件处理器。
```

移除事件处理器

从事件移除处理器，调用 `yii\base\Component::off()` 方法。如：

```
1.  // 处理器是全局函数
2.  $foo-&gt;off(Foo::EVENT_HELLO, 'function_name');

4.  // 处理器是对象方法
5.  $foo-&gt;off(Foo::EVENT_HELLO, [$object, 'methodName']);

7.  // 处理器是静态类方法
8.  $foo-&gt;off(Foo::EVENT_HELLO, ['app\\components\\Bar', 'methodName']);

10.  // 处理器是匿名函数
11.  $foo-&gt;off(Foo::EVENT_HELLO, $anonymousFunction);
```

注意当匿名函数附加到事件后一般不要尝试移除匿名函数，除非你在某处存储了它。以上示例中，假设匿名函数存储为变量 `$anonymousFunction` 。

移除事件的全部处理器，简单调用 `yii\base\Component::off()` 即可，不需要第二个参数：

```
1.  $foo-&gt;off(Foo::EVENT_HELLO);
```

类级别的事件处理器

以上部分，我们叙述了在实例级别如何附加处理器到事件。有时想要一个类的所有实例而不是一个指定的实例都响应一个被触发的事件，并不是一个个附加事件处理器到每个实例，而是通过调用静态方法 `yii\base\Event::on()` 在类级别附加处理器。

例如，活动记录对象要在每次往数据库新增一条新记录时触发一个 `yii\base\ActiveRecord::EVENT_AFTER_INSERT` 事件。要追踪每个活动记录对象的新增记录完成情况，应如下写代码：

```
1.  use Yii;
2.  use yii\\base\\Event;
3.  use yii\\db\\ActiveRecord;

5.  Event::on(ActiveRecord::className(), ActiveRecord::EVENT_AFTER_INSERT, function ($eve
6.  Yii::trace(get_class($event-&gt;sender) . ' is inserted');
7.  });
```

每当 `yii\base\ActiveRecord|ActiveRecord` 或其子类的实例触发 `yii\base\ActiveRecord::EVENT_AFTER_INSERT|EVENT_AFTER_INSERT` 事件时，这个事件处理器都会执行。在这个处理器中，可以通过 `$event->sender` 获取触发事件的对象。

当对象触发事件时，它首先调用实例级别的处理器，然后才会调用类级别处理器。

可调用静态方法 `yii\base\Event::trigger()` 来触发一个类级别事件。类级别事件不与特定对象相关联。因此，它只会引起类级别事件处理器的调用。如：

```
1.  use yii\\base\\Event;

3.  Event::on(Foo::className(), Foo::EVENT_HELLO, function ($event) {
4.  echo $event-&gt;sender;  // 显示 "app\\models\\Foo"
5.  });

7.  Event::trigger(Foo::className(), Foo::EVENT_HELLO);
```

注意这种情况下 `$event->sender` 指向触发事件的类名而不是对象实例。

> 注意：因为类级别的处理器响应类和其子类的所有实例触发的事件，必须谨慎使用，尤其是底层的基类，如 `yii\base\Object`。

移除类级别的事件处理器只需调用 `yii\base\Event::off()`，如：

```
1.  // 移除 $handler
2.  Event::off(Foo::className(), Foo::EVENT_HELLO, $handler);

4.  // 移除 Foo::EVENT_HELLO 事件的全部处理器
5.  Event::off(Foo::className(), Foo::EVENT_HELLO);
```

全局事件

所谓全局事件实际上是一个基于以上叙述的事件机制的戏法。它需要一个全局可访问的单例，如应用实例。

事件触发者不调用其自身的 `trigger()` 方法，而是调用单例的 `trigger()` 方法来触发全局事件。类似地，事件处理器被附加到单例的事件。如：

```
1.  use Yii;
2.  use yii\\base\\Event;
3.  use app\\components\\Foo;

5.  Yii::$app-&gt;on('bar', function ($event) {
6.  echo get_class($event-&gt;sender);  // 显示 "app\\components\\Foo"
7.  });

9.  Yii::$app-&gt;trigger('bar', new Event(['sender' =&gt; new Foo]));
```

全局事件的一个好处是当附加处理器到一个对象要触发的事件时，不需要产生该对象。相反，处理器附加和事件触发都通过单例(如应用实例)完成。

然而，因为全局事件的命名空间由各方共享，应合理命名全局事件，如引入一些命名空间(例："frontend.mail.sent", "backend.mail.sent")。

行为 - Yii2教程

行为

行为是 `yii\base\Behavior` 或其子类的实例。行为，也称为 mixins，可以无须改变类继承关系即可增强一个已有的 `yii\base\Component|组件` 类功能。当行为附加到组件后，它将“注入”它的方法和属性到组件，然后可以像访问组件内定义的方法和属性一样访问它们。此外，行为通过组件能响应被触发的事件，从而自定义或调整组件正常执行的代码。

定义行为

要定义行为，通过继承 `yii\base\Behavior` 或其子类来建立一个类。如：

```
1.  namespace app\\components;

3.  use yii\\base\\Model;
4.  use yii\\base\\Behavior;

6.  class MyBehavior extends Behavior
7.  {
8.  public $prop1;

10.  private $_prop2;

12.  public function getProp2()
13.  {
14.  return $this-&gt;_prop2;
15.  }

17.  public function setProp2($value)
18.  {
19.  $this-&gt;_prop2 = $value;
20.  }

22.  public function foo()
23.  {
24.  // ...
25.  }
26.  }
```

以上代码定义了行为类 `app\components\MyBehavior` 并为要附加行为的组件提供了两个属性 `prop1` 、 `prop2` 和一个方法 `foo()` 。注意属性 `prop2` 是通过 getter `getProp2()` 和 setter `setProp2()` 定义的。能这样用是因为 `yii\base\Object` 是 `yii\base\Behavior` 的祖先类，此祖先类支持用 getter 和 setter 方法定义属性

> 提示：在行为内部可以通过 `yii\base\Behavior::owner` 属性访问行为已附加的组件。

处理事件

如果要让行为响应对应组件的事件触发，就应覆写 `yii\base\Behavior::events()` 方法，如：

```
1.  namespace app\\components;

3.  use yii\\db\\ActiveRecord;
4.  use yii\\base\\Behavior;

6.  class MyBehavior extends Behavior
7.  {
8.  // 其它代码

10.  public function events()
11.  {
12.  return [
13.  ActiveRecord::EVENT_BEFORE_VALIDATE =&gt; 'beforeValidate',
14.  ];
15.  }

17.  public function beforeValidate($event)
18.  {
19.  // 处理器方法逻辑
20.  }
21.  }
```

`yii\base\Behavior::events()|events()` 方法返回事件列表和相应的处理器。上例声明了 `yii\db\ActiveRecord::EVENT_BEFORE_VALIDATE|EVENT_BEFORE_VALIDATE` 事件和它的处理器 `beforeValidate()` 。当指定一个事件处理器时，要使用以下格式之一：

- 指向行为类的方法名的字符串，如上例所示；
- 对象或类名和方法名的数组，如 [$object, 'methodName']；
- 匿名方法。

处理器的格式如下，其中 $event 指向事件参数。关于事件的更多细节请参考事件：

```
1.  function ($event) {
2.  }
```

附加行为

可以静态或动态地附加行为到 `yii\base\Component|组件` 。前者在实践中更常见。

要静态附加行为，覆写行为要附加的组件类的 `yii\base\Component::behaviors()|behaviors()` 方法即可。 `yii\base\Component::behaviors()|behaviors()` 方法应该返回行为配置列表。每个行为配置可以是行为类名也可以是配置数组。如：

```
1.  namespace app\\models;

3.  use yii\\db\\ActiveRecord;
4.  use app\\components\\MyBehavior;

6.  class User extends ActiveRecord
7.  {
8.  public function behaviors()
9.  {
10.  return [
11.  // 匿名行为，只有行为类名
12.  MyBehavior::className(),

14.  // 命名行为，只有行为类名
15.  'myBehavior2' =&gt; MyBehavior::className(),

17.  // 匿名行为，配置数组
18.  [
19.  'class' =&gt; MyBehavior::className(),
20.  'prop1' =&gt; 'value1',
21.  'prop2' =&gt; 'value2',
22.  ],

24.  // 命名行为，配置数组
25.  'myBehavior4' =&gt; [
26.  'class' =&gt; MyBehavior::className(),
27.  'prop1' =&gt; 'value1',
28.  'prop2' =&gt; 'value2',
29.  ]
30.  ];
31.  }
32.  }
```

通过指定行为配置数组相应的键可以给行为关联一个名称。这种行为称为命名行为。上例中，有两个命名行为： `myBehavior2` 和 `myBehavior4` 。如果行为没有指定名称就是匿名行为。

要动态附加行为，在对应组件里调用 `yii\base\Component::attachBehavior()` 方法即可，如：

```
1.  use app\\components\\MyBehavior;

3.  // 附加行为对象
4.  $component-&gt;attachBehavior('myBehavior1', new MyBehavior);

6.  // 附加行为类
7.  $component-&gt;attachBehavior('myBehavior2', MyBehavior::className());

9.  // 附加配置数组
10.  $component-&gt;attachBehavior('myBehavior3', [
11.  'class' =&gt; MyBehavior::className(),
12.  'prop1' =&gt; 'value1',
13.  'prop2' =&gt; 'value2',
14.  ]);
```

可以通过 `yii\base\Component::attachBehaviors()` 方法一次附加多个行为：

```
1.  $component-&gt;attachBehaviors([
2.  'myBehavior1' =&gt; new MyBehavior,  // 命名行为
3.  MyBehavior::className(),          // 匿名行为
4.  ]);
```

还可以通过配置去附加行为：

```
1.  [
2.  'as myBehavior2' =&gt; MyBehavior::className(),

4.  'as myBehavior3' =&gt; [
5.  'class' =&gt; MyBehavior::className(),
6.  'prop1' =&gt; 'value1',
7.  'prop2' =&gt; 'value2',
8.  ],
9.  ]
```

详情请参考配置章节。

使用行为

使用行为，必须像前文描述的一样先把它附加到 `yii\base\Component|component` 类或其子类。一旦行为附加到组件，就可以直接使用它。

行为附加到组件后，可以通过组件访问一个行为的公共成员变量或 getter 和 setter 方法定义的属性：

```
1.  // "prop1" 是定义在行为类的属性
2.  echo $component-&gt;prop1;
3.  $component-&gt;prop1 = $value;
```

类似地也可以调用行为的*公共方法：

```
1.  // bar() 是定义在行为类的公共方法
2.  $component-&gt;bar();
```

如你所见，尽管 `$component` 未定义 `prop1` 和 `bar()` ，它们用起来也像组件自己定义的一样。

如果两个行为都定义了一样的属性或方法，并且它们都附加到同一个组件，那么首先附加上的行为在属性或方法被访问时有优先权。

附加行为到组件时的命名行为，可以使用这个名称来访问行为对象，如下所示：

```
1.  $behavior = $component-&gt;getBehavior('myBehavior');
```

也能获取附加到这个组件的所有行为：

```
1.  $behaviors = $component-&gt;getBehaviors();
```

移除行为

要移除行为，可以调用 `yii\base\Component::detachBehavior()` 方法用行为相关联的名字实现：

```
1.  $component-&gt;detachBehavior('myBehavior1');
```

也可以移除全部行为：

```
1.  $component-&gt;detachBehaviors();
```

使用 `TimestampBehavior`

最后以 `yii\behaviors\TimestampBehavior` 的讲解来结尾，这个行为支持在 `yii\db\ActiveRecord|Active Record` 存储时自动更新它的时间戳属性。

首先，附加这个行为到计划使用该行为的 `yii\db\ActiveRecord|Active Record` 类：

```
1.  namespace app\\models\\User;

3.  use yii\\db\\ActiveRecord;
4.  use yii\\behaviors\\TimestampBehavior;

6.  class User extends ActiveRecord
7.  {
8.  // ...

10.  public function behaviors()
11.  {
12.  return [
13.  [
14.  'class' =&gt; TimestampBehavior::className(),
15.  'attributes' =&gt; [
16.  ActiveRecord::EVENT_BEFORE_INSERT =&gt; ['created_at', 'updated_at'],
17.  ActiveRecord::EVENT_BEFORE_UPDATE =&gt; ['updated_at'],
18.  ],
19.  ],
20.  ];
21.  }
22.  }
```

以上指定的行为数组：

- 当记录插入时，行为将当前时间戳赋值给 created_at 和 updated_at 属性；
- 当记录更新时，行为将当前时间戳赋值给 updated_at 属性。

保存 User 对象，将会发现它的 created_at 和 updated_at 属性自动填充了当前时间戳：

```
$user = new User;
$user-&gt;email = 'test@example.com';
$user-&gt;save();
echo $user-&gt;created_at; // 显示当前时间戳
```

`yii\behaviors\TimestampBehavior|TimestampBehavior` 行为还提供了一个有用的方法 `yii\behaviors\TimestampBehavior::touch()|touch()`，这个方法能将当前时间戳赋值给指定属性并保存到数据库：

```
$user->touch('login_time');
```

与 PHP traits 的比较

尽管行为在 "注入" 属性和方法到主类方面类似于 traits，它们在很多方面却不相同。如上所述，它们各有利弊。它们更像是互补的而不是相互替代。

行为的优势

行为类像普通类支持继承。另一方面，traits 可以视为 PHP 语言支持的复制粘贴功能，它不支持继承。

行为无须修改组件类就可动态附加到组件或移除。要使用 traits，必须修改使用它的类。

行为是可配置的而 traits 不能。

行为以响应事件来自定义组件的代码执行。

当不同行为附加到同一组件产生命名冲突时，这个冲突通过先附加行为的优先权自动解决。而由不同 traits 引发的命名冲突需要通过手工重命名冲突属性或方法来解决。

traits 的优势

traits 比起行为更高效，因为行为是对象，消耗时间和内存。

IDE 对 traits 更友好，因为它们是语言结构。

配置 - Yii2教程

配置

在 Yii 中，创建新对象和初始化已存在对象时广泛使用配置。配置通常包含被创建对象的类名和一组将要赋值给对象属性的初始值。还可能包含一组将被附加到对象事件上的句柄。和一组将被附加到对象上的行为。

以下代码中的配置被用来创建并初始化一个数据库连接：

```
1.  $config = [
2.  'class' =&gt; 'yii\\db\\Connection',
3.  'dsn' =&gt; 'mysql:host=127.0.0.1;dbname=demo',
4.  'username' =&gt; 'root',
5.  'password' =&gt; '',
6.  'charset' =&gt; 'utf8',
7.  ];

9.  $db = Yii::createObject($config);
```

`Yii::createObject()` 方法接受一个配置并根据配置中指定的类名创建对象。对象实例化后，剩余的参数被用来初始化对象的属性，事件处理和行为。

对于已存在的对象，可以使用 `Yii::configure()` 方法根据配置去初始化其属性，就像这样：

```
1.  Yii::configure($object, $config);
```

请注意，如果配置一个已存在的对象，那么配置数组中不应该包含指定类名的 `class` 元素。

配置的格式

一个配置的格式可以描述为以下形式：

```
1.  [
2.  'class' =&gt; 'ClassName',
3.  'propertyName' =&gt; 'propertyValue',
4.  'on eventName' =&gt; $eventHandler,
5.  'as behaviorName' =&gt; $behaviorConfig,
6.  ]
```

其中

- class 元素指定了将要创建的对象的完全限定类名。
- propertyName 元素指定了对象属性的初始值。键名是属性名，值是该属性对应的初始值。只有公共成员变量以及通过 getter/setter 定义的属性可以被配置。

- on eventName 元素指定了附加到对象[事件](#)上的句柄是什么。请注意，数组的键名由 on 前缀加事件名组成。请参考[事件](#)章节了解事件句柄格式。
- as behaviorName 元素指定了附加到对象的[行为](#)。请注意，数组的键名由 as 前缀加行为名组成。$behaviorConfig 表示创建行为的配置信息，格式与我们现在总体叙述的配置格式一样。

下面是一个配置了初始化属性值，事件句柄和行为的示例：

```
1.  [
2.  'class' =&gt; 'app\\components\\SearchEngine',
3.  'apiKey' =&gt; 'xxxxxxxx',
4.  'on search' =&gt; function ($event) {
5.  Yii::info("搜索的关键词： " . $event-&gt;keyword);
6.  },
7.  'as indexer' =&gt; [
8.  'class' =&gt; 'app\\components\\IndexerBehavior',
9.  // ... 初始化属性值 ...
10.  ],
11.  ]
```

使用配置

Yii 中的配置可以用在很多场景。本章开头我们展示了如何使用 `Yii::creatObject()` 根据配置信息创建对象。本小节将介绍配置的两种主要用法 —— 配置应用与配置小部件。

应用的配置

应用的配置可能是最复杂的配置之一。因为 `yii\web\Application|application` 类拥有很多可配置的属性和事件。更重要的是它的 `yii\web\Application::components|components` 属性可以接收配置数组并通过应用注册为组件。以下是一个针对基础应用模板的应用配置概要：

```
1.  $config = [
2.  'id' =&gt; 'basic',
3.  'basePath' =&gt; dirname(__DIR__),
4.  'extensions' =&gt; require(__DIR__ . '/../vendor/yiisoft/extensions.php'),
5.  'components' =&gt; [
6.  'cache' =&gt; [
7.  'class' =&gt; 'yii\\caching\\FileCache',
8.  ],
9.  'mailer' =&gt; [
10.  'class' =&gt; 'yii\\swiftmailer\\Mailer',
11.  ],
12.  'log' =&gt; [
13.  'class' =&gt; 'yii\\log\\Dispatcher',
14.  'traceLevel' =&gt; YII_DEBUG ? 3 : 0,
15.  'targets' =&gt; [
16.  [
17.  'class' =&gt; 'yii\\log\\FileTarget',
18.  ],
19.  ],
20.  ],
21.  'db' =&gt; [
22.  'class' =&gt; 'yii\\db\\Connection',
23.  'dsn' =&gt; 'mysql:host=localhost;dbname=stay2',
24.  'username' =&gt; 'root',
25.  'password' =&gt; '',
26.  'charset' =&gt; 'utf8',
27.  ],
28.  ],
29.  ];
```

配置中没有 `class` 键的原因是这段配置应用在下面的入口脚本中，类名已经指定了。

```
1.  (new yii\\web\\Application($config))-&gt;run();
```

更多关于应用 `components` 属性配置的信息可以查阅应用以及服务定位器章节。

小部件的配置

使用小部件时，常常需要配置以便自定义其属性。 `yii\base\Widget::widget()` 和 `yii\base\Widget::beginWidget()` 方法都可以用来创建小部件。它们可以接受配置数组：

```
1.  use yii\\widgets\\Menu;

3.  echo Menu::widget([
4.  'activateItems' =&gt; false,
5.  'items' =&gt; [
6.  ['label' =&gt; 'Home', 'url' =&gt; ['site/index'`,
7.  ['label' =&gt; 'Products', 'url' =&gt; ['product/index'`,
8.  ['label' =&gt; 'Login', 'url' =&gt; ['site/login'], 'visible' =&gt; Yii::$app-&gt;use
9.  ],
10.  ]);
```

上述代码创建了一个小部件 `Menu` 并将其 `activateItems` 属性初始化为 false。 `item` 属性也配置成了将要显示的菜单条目。

请注意，代码中已经给出了类名 `yii\widgets\Menu` ，配置数组不应该再包含 `class` 键。

配置文件

当配置的内容十分复杂，通用做法是将其存储在一或多个 PHP 文件中，这些文件被称为配置文件。一个配置文件返回的是 PHP 数组。例如，像这样把应用配置信息存储在名为 `web.php` 的文件中：

```
1.  return [
2.  'id' =&gt; 'basic',
3.  'basePath' =&gt; dirname(__DIR__),
4.  'extensions' =&gt; require(__DIR__ . '/../vendor/yiisoft/extensions.php'),
5.  'components' =&gt; require(__DIR__ . '/components.php'),
6.  ];
```

鉴于 `components` 配置也很复杂，上述代码把它们存储在单独的 `components.php` 文件中，并且包含在 `web.php` 里。 `components.php` 的内容如下：

```
1.  return [
2.  'cache' =&gt; [
3.  'class' =&gt; 'yii\\caching\\FileCache',
4.  ],
5.  'mailer' =&gt; [
6.  'class' =&gt; 'yii\\swiftmailer\\Mailer',
7.  ],
8.  'log' =&gt; [
9.  'class' =&gt; 'yii\\log\\Dispatcher',
10.  'traceLevel' =&gt; YII_DEBUG ? 3 : 0,
11.  'targets' =&gt; [
12.  [
13.  'class' =&gt; 'yii\\log\\FileTarget',
14.  ],
15.  ],
16.  ],
17.  'db' =&gt; [
18.  'class' =&gt; 'yii\\db\\Connection',
19.  'dsn' =&gt; 'mysql:host=localhost;dbname=stay2',
20.  'username' =&gt; 'root',
21.  'password' =&gt; '',
22.  'charset' =&gt; 'utf8',
23.  ],
24.  ];
```

仅仅需要 “require”，就可以取得一个配置文件的配置内容，像这样：

```
1.  $config = require('path/to/web.php');
2.  (new yii\\web\\Application($config))-&gt;run();
```

默认配置

`Yii::createObject()` 方法基于依赖注入容器实现。使用 `Yii::creatObject()` 创建对象时，可以附加一系列默认配置到指定类的任何实例。默认配置还可以在入口脚本中调用 `Yii::$container->set()` 来定义。

例如，如果你想自定义 `yii\widgets\LinkPager` 小部件，以便让分页器最多只显示 5 个翻页按钮(默认是 10 个)，你可以用下述代码实现：

```
1.  \\Yii::$container-&gt;set('yii\\widgets\\LinkPager', [
2.  'maxButtonCount' =&gt; 5,
3.  ]);
```

不使用默认配置的话，你就得在任何使用分页器的地方，都配置 `maxButtonCount` 的值。

环境常量

配置经常要随着应用运行的不同环境更改。例如在开发环境中，你可能使用名为 `mydb_dev` 的数据库，而生产环境则使用 `mydb_prod` 数据库。为了便于切换使用环境，Yii 提供了一个定义在入口脚本中的 `YII_ENV` 常量。如下：

```
1.  defined('YII_ENV') or define('YII_ENV', 'dev');
```

你可以把 `YII_ENV` 定义成以下任何一种值：

- prod：生产环境。常量 YII_ENV_PROD 将被看作 true。如果你没修改过，这就是 YII_ENV 的默认值。
- dev：开发环境。常量 YII_ENV_DEV 将被看作 true。
- test：测试环境。常量 YII_ENV_TEST 将被看作 true。

有了这些环境常量，你就可以根据当下应用运行环境的不同，进行差异化配置。例如，应用可以包含下述代码只在开发环境中开启调试工具。

```
1.  $config = [...];

3.  if (YII_ENV_DEV) {
4.  // 根据 `dev` 环境进行的配置调整
5.  $config['bootstrap'][] = 'debug';
6.  $config['modules']['debug'] = 'yii\\debug\\Module';
7.  }

9.  return $config;
```

别名（Aliases） - Yii2教程

别名(Aliases)

别名用来表示文件路径和 URL，这样就避免了在代码中硬编码一些绝对路径和 URL。一个别名必须以 `@` 字符开头，以区别于传统的文件路径和 URL。Yii 预定义了大量可用的别名。例如，别名 `@yii` 指的是 Yii 框架本身的安装目录，而 `@web` 表示的是当前运行应用的根 URL。

定义别名

你可以调用 `Yii::setAlias()` 来给文件路径或 URL 定义别名：

```
1.  // 文件路径的别名
2.  Yii::setAlias('@foo', '/path/to/foo');

4.  // URL 的别名
5.  Yii::setAlias('@bar', 'http://www.example.com');
```

> 注意：别名所指向的文件路径或 URL 不一定是真实存在的文件或资源。

可以通过在一个别名后面加斜杠 `/` 和一至多个路径分段生成新别名(无需调用 `Yii::setAlias()`)。我们把通过 `Yii::setAlias()` 定义的别名称为根别名，而用他们衍生出去的别名成为衍生别名。例如，`@foo` 就是跟别名，而 `@foo/bar/file.php` 是一个衍生别名。

你还可以用别名去定义新别名(根别名与衍生别名均可)：

```
1.  Yii::setAlias('@foobar', '@foo/bar');
```

根别名通常在引导阶段定义。比如你可以在入口脚本里调用 `Yii::setAlias()`。为了方便起见，应用提供了一个名为 `aliases` 的可写属性，你可以在应用配置中设置它，就像这样：

```
1.  return [
2.  // ...
3.  'aliases' =&gt; [
4.  '@foo' =&gt; '/path/to/foo',
5.  '@bar' =&gt; 'http://www.example.com',
6.  ],
7.  ];
```

解析别名

你可以调用 `Yii::getAlias()` 命令来解析根别名到对应的文件路径或 URL。同样的页面也可以用于解析衍生别名。例如：

```
1.  echo Yii::getAlias('@foo');               // 输出：/path/to/foo
2.  echo Yii::getAlias('@bar');               // 输出：http://www.example.com
3.  echo Yii::getAlias('@foo/bar/file.php');  // 输出：/path/to/foo/bar/file.php
```

由衍生别名所解析出的文件路径和 URL 是通过替换掉衍生别名中的根别名部分得到的。

> 注意：`Yii::getAlias()` 并不检查结果路径/URL 所指向的资源是否真实存在。

根别名可能也会包含斜杠 `/` 。 `Yii::getAlias()` 足够智能到判断一个别名中的哪部分是根别名，因此能正确解析文件路径/URL。例如：

```
1.  Yii::setAlias('@foo', '/path/to/foo');
2.  Yii::setAlias('@foo/bar', '/path2/bar');
3.  echo Yii::getAlias('@foo/test/file.php');  // 输出：/path/to/foo/test/file.php
4.  echo Yii::getAlias('@foo/bar/file.php');   // 输出：/path2/bar/file.php
```

若 `@foo/bar` 未被定义为根别名，最后一行语句会显示为 `/path/to/foo/bar/file.php` 。

使用别名

别名在 Yii 的很多地方都会被正确识别，无需调用 `Yii::getAlias()` 来把它们转换为路径/URL。例如， `yii\caching\FileCache::cachePath` 能同时接受文件路径或是指向文件路径的别名，因为通过 `@` 前缀能区分它们。

```
1.  use yii\\caching\\FileCache;

3.  $cache = new FileCache([
4.  'cachePath' =&gt; '@runtime/cache',
5.  ]);
```

请关注 API 文档了解特定属性或方法参数是否支持别名。

预定义的别名

Yii 预定义了一系列别名来简化常用路径和 URL的使用：

- @yii - BaseYii.php 文件所在的目录(也被称为框架安装目录)
- @app - 当前运行的应用 `yii\base\Application::basePath|根路径(base path)`
- @runtime - 当前运行的应用的 `yii\base\Application::runtimePath|运行环境(runtime)路径`
- @vendor - `yii\base\Application::vendorPath|Composer 供应商目录`
- @webroot - 当前运行应用的 Web 入口目录
- @web - 当前运行应用的根 URL

`@yii` 别名是在入口脚本里包含 `Yii.php` 文件时定义的，其他的别名都是在配置应用的时候，于应用的构造方法内定义的。

扩展的别名

每一个通过 Composer 安装的 扩展 都自动添加了一个别名。该别名会以该扩展在 `composer.json` 文件中所声明的根命名空间为名，且他直接代指该包的根目录。例如，如果你安装有 `yiisoft/yii2-jui` 扩展，会自动得到 `@yii/jui` 别名，它定义于引导启动阶段：

```
1.  Yii::setAlias('@yii/jui', 'VendorPath/yiisoft/yii2-jui');
```

服务定位器 - Yii2教程

服务定位器

服务定位器是一个了解如何提供各种应用所需的服务(或组件)的对象。在服务定位器中，每个组件都只有一个单独的实例，并通过ID 唯一地标识。用这个 ID 就能从服务定位器中得到这个组件。

在 Yii 中，服务定位器是 `yii\di\ServiceLocator` 或其子类的一个实例。

最常用的服务定位器是**application(应用)**对象，可以通过 `\Yii::$app` 访问。它所提供的服务被称为**application components(应用组件)**，比如：`request`、`response`、`urlManager` 组件。可以通过服务定位器所提供的功能，非常容易地配置这些组件，或甚至是用你自己的实现替换掉他们。

除了 application 对象，每个模块对象本身也是一个服务定位器。

要使用服务定位器，第一步是要注册相关组件。组件可以通过 `yii\di\ServiceLocator::set()` 方法进行注册。以下的方法展示了注册组件的不同方法：

```
1.  use yii\\di\\ServiceLocator;
2.  use yii\\caching\\FileCache;

4.  $locator = new ServiceLocator;

6.  // 通过一个可用于创建该组件的类名，注册 "cache" (缓存)组件。
7.  $locator-&gt;set('cache', 'yii\\caching\\ApcCache');

9.  // 通过一个可用于创建该组件的配置数组，注册 "db" (数据库)组件。
10.  $locator-&gt;set('db', [
11.  'class' =&gt; 'yii\\db\\Connection',
12.  'dsn' =&gt; 'mysql:host=localhost;dbname=demo',
13.  'username' =&gt; 'root',
14.  'password' =&gt; '',
15.  ]);

17.  // 通过一个能返回该组件的匿名函数，注册 "search" 组件。
18.  $locator-&gt;set('search', function () {
19.  return new app\\components\\SolrService;
20.  });

22.  // 用组件注册 "pageCache" 组件
23.  $locator-&gt;set('pageCache', new FileCache);
```

一旦组件被注册成功，你可以任选以下两种方式之一，通过它的 ID 访问它：

```
1.  $cache = $locator-&gt;get('cache');
2.  // 或者
3.  $cache = $locator-&gt;cache;
```

如上所示， `yii\di\ServiceLocator` 允许通过组件 ID 像访问一个属性值那样访问一个组件。当你第一次访问某组件时， `yii\di\ServiceLocator` 会通过该组件的注册信息创建一个该组件的实例，并返回它。之后，如果再次访问，则服务定位器会返回同一个实例。

你可以通过 `yii\di\ServiceLocator::has()` 检查某组件 ID 是否被注册。若你用一个无效的 ID 调用 `yii\di\ServiceLocator::get()` ，则会抛出一个异常。

因为服务定位器，经常会在创建时附带配置信息，因此我们提供了一个可写的属性，名为 `yii\di\ServiceLocator::setComponents()|components` ，这样就可以配置该属性，或一次性注册多个组件。下面的代码展示了如何用一个配置数组，配置一个应用并注册"db"，"cache" 和 "search" 三个组件：

```
1.  return [
2.  // ...
3.  'components' =&gt; [
4.  'db' =&gt; [
5.  'class' =&gt; 'yii\\db\\Connection',
6.  'dsn' =&gt; 'mysql:host=localhost;dbname=demo',
7.  'username' =&gt; 'root',
8.  'password' =&gt; '',
9.  ],
10.  'cache' =&gt; 'yii\\caching\\ApcCache',
11.  'search' =&gt; function () {
12.  return new app\\components\\SolrService;
13.  },
14.  ],
15.  ];
```

依赖注入容器 - Yii2教程

依赖注入容器

依赖注入(Dependency Injection，DI)容器就是一个对象，它知道怎样初始化并配置对象及其依赖的所有对象。Martin 的文章 已经解释了 DI 容器为什么很有用。这里我们主要讲解 Yii 提供的 DI 容器的使用方法。

依赖注入

Yii 通过 `yii\di\Container` 类提供 DI 容器特性。它支持如下几种类型的依赖注入：

- 构造方法注入;
- Setter 和属性注入;
- PHP 回调注入.

构造方法注入

在参数类型提示的帮助下，DI 容器实现了构造方法注入。当容器被用于创建一个新对象时，类型提示会告诉它要依赖什么类或接口。容器会尝试获取它所依赖的类或接口的实例，然后通过构造器将其注入新的对象。例如：

```
1.  class Foo
2.  {
3.  public function __construct(Bar $bar)
4.  {
5.  }
6.  }

8.  $foo = $container-&gt;get('Foo');
9.  // 上面的代码等价于：
10.  $bar = new Bar;
11.  $foo = new Foo($bar);
```

Setter 和属性注入

Setter 和属性注入是通过配置提供支持的。当注册一个依赖或创建一个新对象时，你可以提供一个配置，该配置会提供给容器用于通过相应的 Setter 或属性注入依赖。例如：

```
1.  use yii\\base\\Object;

3.  class Foo extends Object
4.  {
5.  public $bar;

7.  private $_qux;

9.  public function getQux()
10.  {
11.  return $this-&gt;_qux;
12.  }

14.  public function setQux(Qux $qux)
15.  {
16.  $this-&gt;_qux = $qux;
17.  }
18.  }

20.  $container-&gt;get('Foo', [], [
21.  'bar' =&gt; $container-&gt;get('Bar'),
22.  'qux' =&gt; $container-&gt;get('Qux'),
23.  ]);
```

PHP 回调注入

这种情况下，容器将使用一个注册过的 PHP 回调创建一个类的新实例。回调负责解决依赖并将其恰当地注入新创建的对象。例如：

```
1.  $container-&gt;set('Foo', function () {
2.  return new Foo(new Bar);
3.  });

5.  $foo = $container-&gt;get('Foo');
```

注册依赖关系

可以用 `yii\di\Container::set()` 注册依赖关系。注册会用到一个依赖关系名称和一个依赖关系的定义。依赖关系名称可以是一个类名，一个接口名或一个别名。依赖关系的定义可以是一个类名，一个配置数组，或者一个 PHP 回调。

```
1.  $container = new \\yii\\di\\Container;

3.  // 注册一个同类名一样的依赖关系，这个可以省略。
4.  $container-&gt;set('yii\\db\\Connection');

6.  // 注册一个接口
7.  // 当一个类依赖这个接口时，相应的类会被初始化作为依赖对象。
8.  $container-&gt;set('yii\\mail\\MailInterface', 'yii\\swiftmailer\\Mailer');

10.  // 注册一个别名。
11.  // 你可以使用 $container-&gt;get('foo') 创建一个 Connection 实例
12.  $container-&gt;set('foo', 'yii\\db\\Connection');

14.  // 通过配置注册一个类
15.  // 通过 get() 初始化时，配置将会被使用。
16.  $container-&gt;set('yii\\db\\Connection', [
17.  'dsn' =&gt; 'mysql:host=127.0.0.1;dbname=demo',
18.  'username' =&gt; 'root',
19.  'password' =&gt; '',
20.  'charset' =&gt; 'utf8',
21.  ]);

23.  // 通过类的配置注册一个别名
24.  // 这种情况下，需要通过一个 “class” 元素指定这个类
25.  $container-&gt;set('db', [
26.  'class' =&gt; 'yii\\db\\Connection',
27.  'dsn' =&gt; 'mysql:host=127.0.0.1;dbname=demo',
28.  'username' =&gt; 'root',
29.  'password' =&gt; '',
30.  'charset' =&gt; 'utf8',
31.  ]);

33.  // 注册一个 PHP 回调
34.  // 每次调用 $container-&gt;get('db') 时，回调函数都会被执行。
35.  $container-&gt;set('db', function ($container, $params, $config) {
36.  return new \\yii\\db\\Connection($config);
37.  });

39.  // 注册一个组件实例
40.  // $container-&gt;get('pageCache') 每次被调用时都会返回同一个实例。
41.  $container-&gt;set('pageCache', new FileCache);
```

> Tip: 如果依赖关系名称和依赖关系的定义相同，则不需要通过 DI 容器注册该依赖关系。

通过 `set()` 注册的依赖关系，在每次使用时都会产生一个新实例。可以使用 `yii\di\Container::setSingleton()` 注册一个单例的依赖关系：

```
1.  $container-&gt;setSingleton('yii\\db\\Connection', [
2.  'dsn' =&gt; 'mysql:host=127.0.0.1;dbname=demo',
3.  'username' =&gt; 'root',
4.  'password' =&gt; '',
5.  'charset' =&gt; 'utf8',
6.  ]);
```

解决依赖关系

注册依赖关系后，就可以使用 DI 容器创建新对象了。容器会自动解决依赖关系，将依赖实例化并注入新创建的对象。依赖关系的解决是递归的，如果一个依赖关系中还有其他依赖关系，则这些依赖关系都会被自动解决。

可以使用 `yii\di\Container::get()` 创建新的对象。该方法接收一个依赖关系名称，它可以是一个类名，一个接口名或一个别名。依赖关系名或许是通过 `set()` 或 `setSingleton()` 注册的。你可以随意地提供一个类的构造器参数列表和一个configuration 用于配置新创建的对象。例如：

```
1.  // "db" 是前面定义过的一个别名
2.  $db = $container-&gt;get('db');

4.  // 等价于： $engine = new \\app\\components\\SearchEngine($apiKey, ['type' =&gt; 1]);
5.  $engine = $container-&gt;get('app\\components\\SearchEngine', [$apiKey], ['type' =&gt
```

代码背后，DI 容器做了比创建对象多的多的工作。容器首先将检查类的构造方法，找出依赖的类或接口名，然后自动递归解决这些依赖关系。

如下代码展示了一个更复杂的示例。 `UserLister` 类依赖一个实现了 `UserFinderInterface` 接口的对象； `UserFinder` 类实现了这个接口，并依赖于一个 `Connection` 对象。所有这些依赖关系都是通过类构造器参数的类型提示定义的。通过属性依赖关系的注册，DI 容器可以自动解决这些依赖关系并能通过一个简单的 `get('userLister')` 调用创建一个新的 `UserLister` 实例。

```
1.  namespace app\\models;

3.  use yii\\base\\Object;
4.  use yii\\db\\Connection;
5.  use yii\\di\\Container;

7.  interface UserFinderInterface
8.  {
9.  function findUser();
10.  }

12.  class UserFinder extends Object implements UserFinderInterface
13.  {
14.  public $db;

16.  public function __construct(Connection $db, $config = [])
17.  {
18.  $this-&gt;db = $db;
19.  parent::__construct($config);
20.  }

22.  public function findUser()
23.  {
24.  }
25.  }

27.  class UserLister extends Object
28.  {
29.  public $finder;

31.  public function __construct(UserFinderInterface $finder, $config = [])
32.  {
33.  $this-&gt;finder = $finder;
34.  parent::__construct($config);
35.  }
36.  }

38.  $container = new Container;
39.  $container-&gt;set('yii\\db\\Connection', [
40.  'dsn' =&gt; '...',
41.  ]);
42.  $container-&gt;set('app\\models\\UserFinderInterface', [
43.  'class' =&gt; 'app\\models\\UserFinder',
44.  ]);
45.  $container-&gt;set('userLister', 'app\\models\\UserLister');

47.  $lister = $container-&gt;get('userLister');

49.  // 等价于:

51.  $db = new \\yii\\db\\Connection(['dsn' =&gt; '...']);
52.  $finder = new UserFinder($db);
53.  $lister = new UserLister($finder);
```

实践中的运用

当在应用程序的入口脚本中引入 `Yii.php` 文件时，Yii 就创建了一个 DI 容器。这个 DI 容器可以通过 `Yii::$container` 访问。当调用 `Yii::createObject()` 时，此方法实际上会调用这个容器的 `yii\di\Container::get()|get()` 方法创建新对象。如上所述，DI 容器会自动解决依赖关系(如果有)并将其注入新创建的对象中。因为 Yii 在其多数核心代码中都使用了 `Yii::createObject()` 创建新对象，所以你可以通过 `Yii::$container` 全局性地自定义这些对象。

例如，你可以全局性自定义 `yii\widgets\LinkPager` 中分页按钮的默认数量:

```
1.  \\Yii::$container-&gt;set('yii\\widgets\\LinkPager', ['maxButtonCount' =&gt; 5]);
```

这样如果你通过如下代码在一个视图里使用这个挂件，它的 `maxButtonCount` 属性就会被初始化为 5 而不是类中定义的默认值 10。

```
1.  echo \\yii\\widgets\\LinkPager::widget();
```

然而你依然可以覆盖通过 DI 容器设置的值：

```
1.  echo \\yii\\widgets\\LinkPager::widget(['maxButtonCount' =&gt; 20]);
```

另一个例子是借用 DI 容器中自动构造方法注入带来的好处。假设你的控制器类依赖一些其他对象，例如一个旅馆预订服务。你可以通过一个构造器参数声明依赖关系，然后让 DI 容器帮你自动解决这个依赖关系。

```
1.  namespace app\\controllers;

3.  use yii\\web\\Controller;
4.  use app\\components\\BookingInterface;

6.  class HotelController extends Controller
7.  {
8.  protected $bookingService;

10.  public function __construct($id, $module, BookingInterface $bookingService, $config
11.  {
12.  $this-&gt;bookingService = $bookingService;
13.  parent::__construct($id, $module, $config);
14.  }
15.  }
```

如果你从浏览器中访问这个控制器，你将看到一个报错信息，提醒你 `BookingInterface` 无法被实例化。这是因为你需要告诉 DI 容器怎样处理这个依赖关系。

```
1.  \\Yii::$container-&gt;set('app\\components\\BookingInterface', 'app\\components\\Book
```

现在如果你再次访问这个控制器，一个 `app\components\BookingService` 的实例就会被创建并被作为第三个参数注入到控制器的构造器中。

什么时候注册依赖关系

由于依赖关系在创建新对象时需要解决，因此它们的注册应该尽早完成。如下是推荐的实践：

- 如果你是一个应用程序的开发者，你可以在应用程序的入口脚本或者被入口脚本引入的脚本中注册依赖关系。
- 如果你是一个可再分发扩展的开发者，你可以将依赖关系注册到扩展的引导类中。

总结

依赖注入和服务定位器都是流行的设计模式，它们使你可以用充分解耦且更利于测试的风格构建软件。强烈推荐你阅读 Martin 的文章 ，对依赖注入和服务定位器有个更深入的理解。

Yii 在依赖住入(DI)容器之上实现了它的服务定位器。当一个服务定位器尝试创建一个新的对象实例时，它会把调用转发到 DI 容器。后者将会像前文所述那样自动解决依赖关系。